2

AD-A242 092

MENTATION PAGE

Form Approved OPM No. 0704-0188

average 1 hour per response, including the time for reviewing instructions, searching existing data sources gathering and maintaining the data garding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington rts, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Information and Regulatory Affairs, Office of

2. REPORT DATE SEPTEMBER 1991	3. REPORT TYPE AND DATES COVERED Final: 11-13 SEPTEMBER 1991
4. TITLE AND SUBTITLE SIXTH ANNUAL ASEET SYMPOSIUM PROCEEDINGS	5. FUNDING NUMBERS
6. AUTHOR(S)	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) INSTITUTE FOR DEFENSE ANALYSES (IDA) 1801 N. BEAUREGARD ST.	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Ada Joint Program Office The Pentagon, Rm. 3E114 Washington, D.C.	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES	
12a. DISTRIBUTION/AVAILABILITY STATEMENT UNCLASSIFIED - UNLIMITED PUBLIC DISTRIBUTION	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

THE PROCEEDINGS CONSISTS OF THE FOLLOWING PAPERS: UNDERGRADUATE SOFTWARE ENGINEERING COURSES: MEETING THE NEEDS OF INDUSTRY; USING ADA TO TEACH CONCURRENCY; LESSONS LEARNED IN THE ADA TRAINING PROGRAMS AT ROCKWELL; THE ADA APPRENTICE; A SEQUENCE OF FRESHMAN LEVEL INTEGRATED LABORATORY ASSIGNMENTS; A TOOL SUPPORTING PROGRAMMING IN THE LARGE FOR THE INTRODUCTORY SOFTWARE DEVELOPMENT COURSES; A TOP-DOWN TOOLBOX APPROACH TO TEACHING THE ADA PROGRAMMING LANGUAGE; AND USING A LANGUAGE SENSITIVE EDITOR AND ADA IN COMPUTER SCIENCE I-II

14. SUBJECT TERMS ADA, EDUCATION, TRAINING, CONCURRENCY, SOFTWARE ENGINEERING, COMPUTER PROGRAMMING LANGUAGE			15. NUMBER OF PAGES 108 16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT NONE

NSN 7540-01-280-550

Standard Form 298, (Rev. 2-89)
Prescribed by ANSI Std. 239-128

Sixth Annual ASEET Symposium Proceedings

91-14344

Alexandria, Virginia
11-13 September 1991

A forum sponsored by the Ada Software Engineering Education and Training (ASEET) Team

146

91-5400

PROCEEDINGS OF THE SIXTH ANNUAL ADA SOFTWARE ENGINEERING EDUCATION AND TRAINING SYMPOSIUM

Sponsored by:

Ada Software Engineering Education and Training Team

and

Ada Joint Program Office

Institute for Defense Analyses
Alexandria, VA

ASEET TEAM MEMBERSHIP
1 September 1991

Eugene Bingue
1942 Comm Sq/LGN
Homestead AFB, FL 33039-6346

1LT Sandra Chandler
Software Engineering Training Branch
3390 TCHTG/TTMKPP
Keesler AFB, Mississippi 39534-5000

Captain David A. Cook
Department of Computer Science
U.S. Air Force Academy, CO 80840

Major Tom Croak
HQ USAF/SCXS
Washington, D.C. 20330-519

Major Chris Demery
Ada Joint Program Office
Room 3E114
The Pentagon
Washington, D.C. 20301-3081

Mr. Leslie W. Dupaix
USAF Software Technology Support Center
OO-ALC/TISAC
Hill AFB, Utah 84056

Major Drew HamiltonAttn: ATZH-SSW
Software Engineering Branch
USA Computer Science School
Fort Gordon, GA 30905

Captain Mike Helsabeck
1500 CSGP/EN
Scott AFB, IL 62225

Captain Dan Herod
HQ ATC/SCDBT
Randolph AFB, Texas 78150

Captain Joyce Jenkins
HQ SCCC/IIATT
Offutt AFB, NE 68123-5001

Ms. Pam Kimminau
9800 Savage Road
Ft. Meade, MD 20755-6000

LtCol Pat Lawlis
AFIT/ENG
Wright Patterson AFB, Ohio 45433

Ms. Cathy McDonald
nstitute for Defense Analyses
1801 N. Beauregard Street
Alexandria, VA 22311

LCdr Lindy Moran
PACOPSUPPFAC
Box 9
Pearl Harbor, HI 96860-7150

Major John Myer
Computer Science
U.S. Naval Academy
Annapolis, MD 21401

Prof. E.K. Park
Assist. Professor
Computer Science (M.S. 9F)
U.S. Naval Academy
Annapolis, MD 21402

Captain Michael Simpson
CTS/RMC
Lackland AFB, Texas 78236-5000

Major J. J. Spegele
USMC
Marine Corps Support Activity
1500 East 95th Street
ATT: Code TY
Kansas City, Missouri 64197-0501)

Major David Umphress
HQ SCCC/IIAT
Offutt AFB, NE 68113

Capt David Vega
Det 1, 3390 TCHTG
45335 Vintage Park Plaza
Sterling, VA 22170-6701

Capt Pat Wicker
SCCC/XPTSE
Offitt AFB
Omaha, NE 68113

TABLE OF CONTENTS

This Page Intentionally Left Blank

Message From The Symposium Chair

Ms. Catherine W. McDonald

It gives me great please to welcome all of you to the Sixth Annual ASEET Symposium. The Team is very excited about this year's symposium and its theme: *Ada as an Educational Tool: The Time is Now*. Over the past 12 months, the Ada Joint Program Office has received numerous requests for training in Ada. Truly, the time is now for all educators to push Ada in their Services, organizations,and universities.

This year, despite the short time span between the Call for Papers and the submission date, we received numerous papers. All the papers were excellent and the final decision of the reviewers was not easy. I hope you find the papers and the panels this year stimulating and thought provoking. Hopefully, the symposium will provide you with the opportunity to interact and exchange ideas with other educators and trainers.

I would also like to take this opportunity to thank all the attendees for their support of the ASEET Team and its activities. If you have any questions about the team, please feel free to ask one of the Team members during the symposium (easily recognized by the green ribbon attached to their name tags).

Again, welcome and thank you.

This Page Intentionally Left Blank

6TH ASEET SYMPOSIUM
11-13 SEPTEMBER, 1991

UNDERGRADUATE SOFTWARE ENGINEERING COURSES: MEETING THE NEEDS OF INDUSTRY

Barbara Hilden
Bruce Johnston

Mathematics Department
University of Wisconsin-Stout
Menomonie, WI 54751

Introduction

Most software development projects undertaken at the undergraduate college level are, by necessity, designed to be accomplished by one or a few students in the course of a single academic term. Moreover, the software is seldom used once the development is completed. This contrasts with the vast majority of industrial software which is complex, developed by large teams of people and must be maintained for several years.

Another contrast with industrial software development is that the requirements for most college software development projects are provided to the student by the faculty. One of the more formidable tasks in industrial software development is the development of the requirements through extensive negotiation and coordination with the end user or customer. These differences result in many computer science graduates having difficulty participating effectively in large scale industrial software development. Since the skills needed to be effective in this environment are primarily acquired on the job, a profound productivity lapse occurs. This shortcoming needs to be addressed at the academic level.

The lack of software engineering principles in early courses is one of the major problems of the Computer Science curriculum as a whole (Werth, 1988). Specifically software engineering principles and skills can not be acquired in a one term Software Engineering class. This is especially true when the students must "unlearn" the development style that was used in their previous classes.

Software Engineering courses have several shortcomings which lead to the overall problem of unskilled Software Engineers. Project teams of three to four students often develop small, throw away software. Most of the requirements for this software are supplied by the instructor and have little or no real-world application. While attempting to get beyond disposable software, most Software Engineering courses fall short. There is little or no configuration management or quality assurance in the projects completed in Software Engineering courses. Additionally, software evaluation is rarely completed adequately, if at all. Finally, students have little idea of the maintenance phase of the software life cycle since this is routinely ignored in Software Engineering courses. Time constraints in a one term Software Engineering course are the biggest contributor to most of these problems.

Possible Steps Toward a Solution

Software Development Studio Environment (Tomayko, 1991)

The software development atmosphere, as suggested by Tomayko, requires students to participate in a 12-16 month software development project conducted in a studio type environment. Students within this environment produce software that is to be delivered to a customer with whom they are in contact. Several faculty members act as an interim review team, as well as coaches for individual students. This 12-16 month commitment allows students sufficient time for a complete pass through the software life cycle but demands considerable amounts of faculty involvement.

This approach was used successfully at Carnegie Mellon University. However, potential problems include the availability of an industrial project and the time and eff t required of the faculty. Faculty of smaller universities are often required to teach a full 12 credit load with little or no release time for additional time commitments, which would be required to implement the Tomayko environment. Finally, the necessary restructuring of student class schedules to accommodate the time commitment is also a concern.

One semester Software Maintenance Course (Engle, et. al 1989)

The one semester Software Maintenance course is based on the use of a predefined software artifact (10,000 - 20,000 lines). Exercises would be included focusing on the topics of configuration management, regression testing, code reviews and stepwise abstraction. The Documented Ada Style Checker (DASC) is available for such use through the SEI Education Program. The package includes the DASC in many forms, including a PC version using the Meridian AdaVantage compiler, student exercises and instructions for both students and instructors. This package can be used during a one semester maintenance course, or parts of the package can be used during a Software Engineering course. Although the maintenance course would be of great help in meeting the industry's need for qualified Software Engineers, it is felt that the magnitude of the project would be difficult for undergraduate students to handle in the typical Computer Science curriculum.

Variations of Existing Software Engineering Courses

One variation of the existing Software Engineering courses would be the incorporation of a 3-4 week maintenance phase at the end of a one semester Software Engineering course. This approach has been used once at the University of Wisconsin-Stout during the Spring 1990 semester. The project was a computerized math quiz bowl system that had to be completed in time for use at a conference during the first week of April. This left over three weeks for evaluation and feedback from the customer and subsequent modifications of the system. Although the student participation in the maintenance activities was instructive, too many other important topics that are usually covered had to be omitted in order to deliver the project on time. Overall, a recommendation of this approach for a one semester project-oriented Software Engineering course can not be given.

The idea of enhancing existing software is, however, well worth some additional thought so a two course series is being implemented. The first semester will be the creation of a software system from the ground up, going through the software life cycle excluding the maintenance phase. The second course will add enhancements to the software written in the first course. One problem to be addressed is that the courses need to be independent, in that students could take the first without immediately following with the second. Careful consideration of this and thorough documentation should alleviate this concern. This method is being adopted for the Fall and Spring semesters of the 1991-92 academic year at the University of Wisconsin-Stout.

More Extensive Use of Industrial Internship Experiences

Many Computer Science programs have found it beneficial for their students to participate in an industrial internship/co-op experience some time during their degree program. Internship experiences have been successfully incorporated into the Software Development concentrations of the Applied Mathematics Degree program at the University of Wisconsin-Stout. Table 1 shows some typical degree programs focusing on sequencing of the Computer Science courses relative to the internship experiences. These case studies illustrate the broad spectrum of degree programs over the past several years.

During the past 10 years, the industrial internship has evolved to become an important component of the degree program at the University of Wisconsin-Stout. While it is not currently a degree requirement, nearly 85% of the students participate in at least one internship. Over half of the students complete two or more such internships. This industrial experience keeps the placement rate for Stout graduates consistently above 95%. However, most students need 9-10 semesters to complete their degree program. In addition, missing part of an academic year while on an internship can cause problems with fall/spring sequence courses.

Perhaps the biggest benefit of internship experience is the dramatic change in the students' attitudes. Most students return to their Computer Science course work with a higher degree of interest and enthusiasm. Participation in a real-world software development project gives a clearer perspective on professional software engineering. Specifically, students understand the need for the following:

- an organized and disciplined approach to software development,
- good written and oral communications skills,
- accurate, complete, and current software documentation.

Although the quality of the software engineering practices at the individual internship sites varies considerably, students come to appreciate the need for an organized and disciplined software engineering methodology.

Internship experiences are useful in several other ways for both students and employers. The students get an opportunity to "test-drive" a job and make sure that this is the right career for them. Specifically, it helps students more precisely define what type of software development they want to pursue for a career. Those students who have had internship experiences in several different areas or for very different types of corporations are typically more specific about what type of software development position they want following graduation. For this reason, the Computer Science faculty generally encourages students to do several different types of internship experiences. However, occasionally an internship experience will convince a student that they are not cut out for a software development career and they subsequently change majors.

Table 1. Case Studies of Student Internship Experiences

CASE STUDY #1 (8 semesters)			
Year	Fall	Spring	Summer
84-85	Computer Science I	Computer Science II	
85-86	Assembly Lang Prog	**IBM-San Jose**	
86-87	Data Structures	Cobol Programming	**IBM-Rochester**
87-88	Computer Organization	Systems Programming Software Engineering	
JOB: IBM, Rochester, MN			

CASE STUDY #2 (9 semesters)			
Year	Fall	Spring	Summer
85-86	Computer Science I	Computer Science II	
86-87	Assembly Lang Prog	Data Structures	**IBM-San Jose**
87-88	**IBM-San Jose**	**IBM-Boca Raton**	
88-89	Computer Organization Computer Graphics	Systems Programming	
89-90	Software Engineering		
JOB: IBM, Gaithersburg, MD			

CASE STUDY #3 (10 semesters)			
Year	Fall	Spring	Summer
84-85	Computer Science I	Computer Science II	
85-86	Assembly Lang Prog Data Structures	**IBM-Owego**	
86-87	Computer Graphics	Image Processing	**Cray Research**
87-88	**Cray Research**	Cobol Programming	
88-89	Computer Organization Software Engineering	Systems Programming	
JOB: XonTech, Van Nuys, CA			

Industrial experiences are also beneficial for employers in several ways. First, they allow employers to test the software development skills and ability of students before offering them a full time position. This is particularly important for students whose professional ability is not always correlated with their college GPA. Most of the internship experiences are long enough that students can become sufficiently involved in a project to do a significant amount of productive work. However, an internship coordinator must be careful to avoid internships that degenerate into clerical type positions. On the other extreme, internships used as a cheap way to get around a company wide hiring freeze for full-time positions should also be avoided.

Undergraduate Degree in Software Engineering (Ford, 1991)

The undergraduate degree in Software Engineering, as described by Ford, involves an engineering approach to the entire curriculum. The proposed curriculum consists of courses which cover many of the same topics that are covered in Computer Science curricula today, but with more of an engineering structure added. A stronger engineering design component is provided in the form of two one-year project courses. This gives students the opportunity for two complete passes through the software life cycle. This proposed curriculum is viewed by the authors of this paper as an important complement to the traditional Computer Science curriculum. However, adoption of this approach will take a significant amount of work since its course structure, while still covering similar topics, is different from most Computer Science curricula.

Application of Suggested Solutions at University of Wisconsin-Stout

Being a small university, University of Wisconsin-Stout Computer Science faculty were able to agree on necessary changes to incorporate a compromise of these suggestions. It was determined that Software Engineering principles, including maintenance, need to be introduced as early as possible into the students' course work. Therefore, adaptation of a Software Engineering structure throughout all Computer Science courses is being implemented. The introduction of the second Software Engineering course will allow for one complete pass through the software life cycle at the academic level thus encompassing portions of Tomayko's studio environment. The continued encouragement of industrial internships addresses the real-world portion of the studio environment while complementing the controlled academic atmosphere. It is not felt that the solution adapted at University of Wisconsin-Stout is the final answer to improved Software Engineering education but it is a step in the right direction.

Summary and Conclusions

In recent years it has become increasingly apparent that Software Engineering is becoming a well defined distinct subset of Computer Science. It is hoped that continued discussions of these ideas and others will eventually lead to strong undergraduate curriculums that meet the Software Engineering needs of industry.

REFERENCES

Ford, Gary, 1991: "The SEI Undergraduate Curriculum in Software Engineering", In Proceedings of the n SIGCSE Technical Symposium on Computer Science Education, SIGCSE Bulletin 23 1 (March 1991). ACM SIGCSE, Washington, D. C., 375-385.

Engle, Charles B., Jr., and Gary Ford, 1990: "Software Maintenance Exercises for a Software Engineering Project Course", In Proceedings of the Fifth Annual Ada Software Engineering Education and Training Symposium, AJPO, 3-9.

Tomayko, James E., "Teaching Software Development in a Studio Environment", In Proceedings of the 22nd SIGCSE Technical Symposium on Computer Science Education, SIGCSE Bulletin 23 1 (March 1991). ACM SIGCSE, Washington, D. C., 300-303

Werth, Laurie Honour, 1988: "Integrating Software Engineering into an Intermediate Programming Class", In Proceedings of the 20th SIGCSE Technical Symposium on Computer Science Education, SIGCSE Bulletin 20 1 (February 1988). ACM SIGCSE, Washington, D. C., 54-58.

Ada and Concurrency in the Classroom

Using Ada To Teach Concurrency

Robert A. Willis Jr.

Hampton University
Department of Computer Science
Hampton, VA 23668

willis@willis.hamptonu.edu

Introduction

This paper discusses experiences found in teaching an upper-level topics course in concurrency. The course builds upon the introduction to concurrency our students are exposed to in Operating Systems I, canvasses a number of concurrent programming models, and looks at important issues concerning concurrent programming. Ada is used extensively throughout the course to simulate symmetric processing and basic synchronization primitives, as a medium to view several concurrent notations, and as a means to perform new and challenging concurrent programming exercises.

Course Structure

A large portion (30%) of the course is concerned with a Survey of Concurrent Processing.[1] [2] We discuss specification notations and synchronization primitives based on shared variables and message passing. A discussion of two or three concurrent programming models usually follows. We also look at one concurrent programming language (in addition to Ada). Finally, we look at some of the limitations of different communication methods, anomaly detection, and methods to analyze concurrent programs.

Why Ada?

Ada is one of the few languages which has safe and general concurrent features. These features are not experimental, they conform to the same consistent philosophy of block-structuring, strong typing, and sound software design principles found throughout Ada, and they are unambiguously specified. These are not features which were grafted onto a programming language, but rather designed as an integral part of Ada. Therefore, their syntax and usage is regular and consistent with all other features found in the language. Since Ada is a general purpose programming language, it is quite easy to integrate concurrent and sequential processing into a program allowing one to develop programs with a rational balance of concurrency as needed.

Ada is particularly well adapted to concurrent program design and development because the modularity of its tasking mechanism supports object oriented design (OOD) quite well. OOD is a tool students have found very helpful in alleviating the complexity that often accompanies concurrent programming.

Ada and Concurrency in the Classroom

Ada as an expository tool

Ada's concurrent features can be used to simulate coroutines quite easily. They can simulate and demonstrate other programming notations features such as the fork/join and cobegin/coend construct. All of the synchronization primitives based on shared variables (busy-waiting, semaphores, conditional critical regions, monitors, and path expressions) can also be simulated.[1] This allows the student to actually see how these constructs work and learn their relative advantages and dis-

1. Simulations can be implemented to various degrees of faithfulness. I recommend implementing only those features which are relevant to a good understanding of the construct. I did not, for instance, implement conditional wait for monitors.

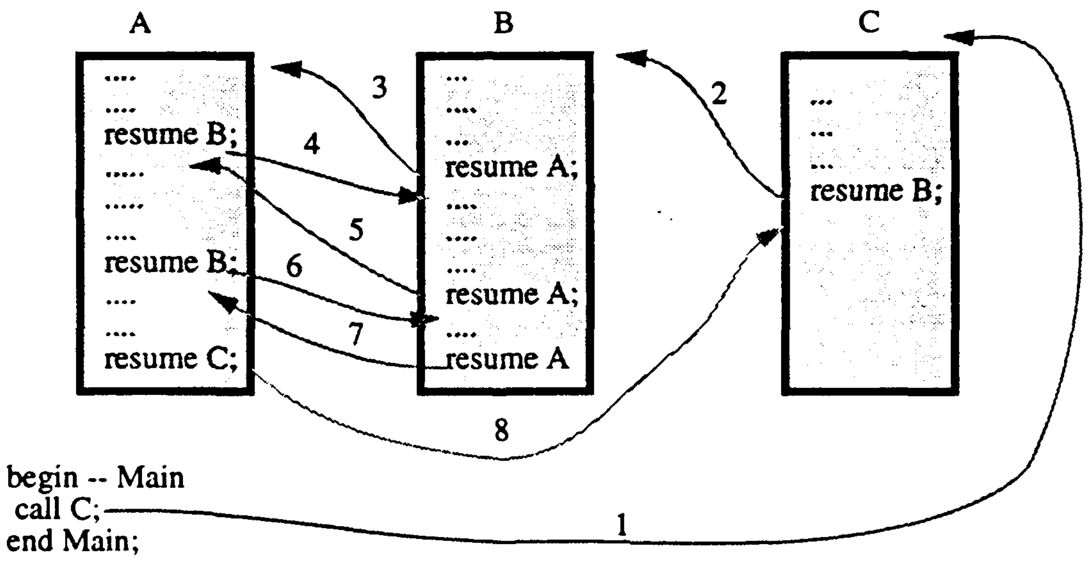
advantages without having to learn different programming languages. Mini-programming exercises can be assigned to illustrate salient features or deficiencies. A sampling of classroom uses will be discussed in the following sections.

Coroutines

Coroutines are the classic example of symmetric concurrency and are included in some languages because of their simplicity and power to provide solutions for a large class of concurrent problems. They are not an effective construct for use in a multiprocessor implementation because only one routine is executing at a time. The programmer must also completely specify all process switching.[2] This last requirement adds to the complexity of a concurrent program and, as such, is not desirable.

Coroutines are a form of subprogram which have multiple entries and exits. Threads of execution are produced by each coroutine explicitly relinquishing control to another coroutine and pausing in its execution until invoked by a resume statement in another coroutine, at which time execution is continued with the statement following the coroutine call. The following figure illustrates the operation of coroutines.

FIGURE 1. Coroutine Operation

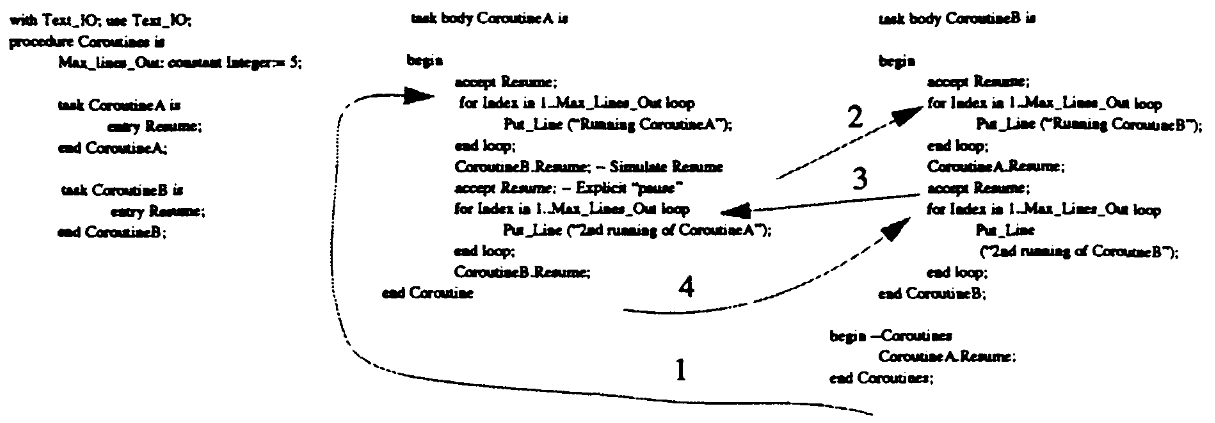


Simulating coroutines in Ada is uncomplicated. A pair of simple rendezvous statements are used to simulate the coroutine resume statement. The pair consists of an entry call to another task (simulated coroutine) immediately followed by an accept statement. The accept statement is used to explicitly halt execution of the calling task thereby simulating the implicit semantics of the resume statement.

The following program serves as an simple example. Each task has a ***resume*** entry declaration which can be used as many times as necessary. Note that, with the exception of the last statement in task CoroutineA, each invocation of another coroutine is in the form of an entry call/accept statement pair simulating true coroutine semantics. This exception is necessary to allow task (and therefore program) termination.

Using this mechanism a number of exercises can be assigned which will afford the student the "essence" of coroutine programming.

FIGURE 2. Coroutine Implementation



```
with Text_IO; use Text_IO;
procedure Coroutines is
        Max_lines_Out: constant Integer:= 5;

        task CoroutineA is
                entry Resume;
        end CoroutineA;

        task CoroutineB is
                entry Resume;
        end CoroutineB;
```

```
task body CoroutineA is

begin
        accept Resume;
        for Index in 1..Max_Lines_Out loop
                Put_Line ("Running CoroutineA");
        end loop;
        CoroutineB.Resume; -- Simulate Resume
        accept Resume; -- Explicit "pause"
        for Index in 1..Max_Lines_Out loop
                Put_Line ("2nd running of CoroutineA");
        end loop;
        CoroutineB.Resume;
end Coroutine
```

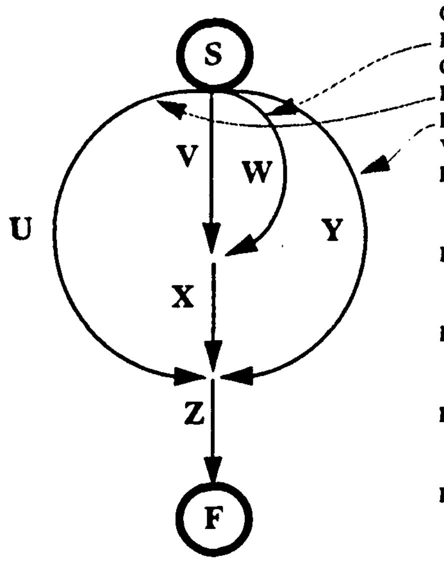
```
task body CoroutineB is

begin
        accept Resume;
        for Index in 1..Max_Lines_Out loop
                Put_Line ("Running CoroutineB");
        end loop;
        CoroutineA.Resume;
        accept Resume;
        for Index in 1..Max_Lines_Out loop
                Put_Line
                 ("2nd running of CoroutineB");
        end loop;
end CoroutineB;

begin --Coroutines
        CoroutineA.Resume;
end Coroutines;
```

Fork/Join statements

Fork statements provide a low level (low level from the standpoint that these statements may be interspersed within program control statements), powerful, and general means of producing two concurrent threads of execution.[5] ***Join*** statements are used to unite two concurrent threads of execution. They are powerful constructs which can be used to model any process flow graph. A process flow graph illustrates the precedence constraints that occur within a concurrent program. [6] The following figure shows a code sequence, its process flow graph, and corresponding fork/join sequence.

FIGURE 3. Fork/Join

```
U:= A * B;
V:= C / B;
W:= E + F;
X:= W + V;
Y:= G ** H;
Z:= X * Y * U;
```



```
Count1:= 2;
Fork Process_2;
Count2:= 3;
Fork Process_4;
Fork Process _5;
V:= C / B; Join Count1, Process_3; Quit;
Process_2: W:= E + F;
        Join Count1, Process_3;
        Quit;
Process_3: X:= W + V;
        Join Count2, Process_6;
        Quit;
Process_4: U:= A * B;
        Join Count2, Process_6;
        Quit;
Process_5: Y:= G ** H;
        Join Count2, Process_6;
        Quit;
Process_6: Z:= X * Y * U;
```

Implementation of the ***Fork*** statement in Ada is in the form of an entry (Fork) in any task(s) with which concurrent execution is required. Implementation of the ***Join*** statement is more involved, it is necessary to declare a task type which contains an parameterized entry (Initialize). The parameters consist of the count and a pointer to the task which will execute next. The count is decremented at each Join rendezvous and control passes to the next task when the count reaches zero. A ***Join*** task must be instantiated for each count variable. The following Ada program simulates the process graph and ***Fork*** and ***Join*** statement sequence given above:

FIGURE 4. Fork/Join Implementation

```
with Text_IO;
procedure Fork_Join is
–
        A: Integer:= 5; – Values hard coded
        B: Integer:= 6;
        C: Integer:= 7;
        E: Integer:= 8;
        F: Integer:= 9;
        G: Integer:= 3;
        H: Integer:= 11;
        U, V, W, X, Y, Z: Integer;

        package Int_IO is new Text_IO.Integer_IO (Integer);
         type Process_Name_Type is (P1, P2, P3, P4, P5, P6); – Used for task Identification

        task type Process is
                entry Initialize (Process_Name: in Process_Name_Type);
                entry Fork;
        end Process;

        type Process_Pointer is access Process;
        Process_1, Process_2, Process_3, Process_4,
        Process_5, Process_6: Process_Pointer;

        task type Join_Task is
 --
 – Initialize receives the Number of forks the counter uses to determine
 – when to branch to the next process, it also receives a pointer to the
 – next process to execute.
                entry Initialize (Number_Of_Forks: in Positive;
                                  Next_Task: Process_Pointer);
                entry Join;
        end Join_Task;

        Count1, Count2: Join_Task;

        task body Join_Task is

                Max_Count: Positive; – Used to keep track of joins
                Next_Process: Process_Pointer;

        begin – Join_Task
                accept Initialize (Number_Of_Forks: in Positive;
                                  Next_Task: Process_Pointer) do
                        Max_Count:= Number_Of_Forks;
                        Next_Process:= Next_Task;
                end Initialize;
                for Counter in 1..Max_Count loop
                        accept Join; -- Simulates Join n times
                end loop;
                Next_Process.Fork; -- Execute next process
        end Join_Task;

        task body Process is

                Process_ID: Process_Name_Type;

        begin – Process
                accept Initialize
                (Process_Name: in Process_Name_Type) do
                        Process_ID:= Process_Name;
                end Initialize;

                accept Fork;

                case Process_ID is
                        when P1 => V:= C / B;
                                        Count1.Join;
                        when P2 => W:= E + F;
                                        Count1.Join;
                        when P3 => X:= W + V;
                                        Count2.Join;
                        when P4 => U:= A * B;
                                        Count2.Join;
                        when P5 => Y:= G ** H;
                                        Count2.Join;
                        when P6 => Z:= X * Y * U;
                                        Text_IO.Put ("The answer is: ");
                                        Int_IO.Put (Z);
                                        Text_IO.New_Line;
                end case;
        end Process;

begin -- Fork_Join
        Process_1:= new Process; – Instantiate each process
        Process_2:= new Process;
        Process_3:= new Process;
        Process_4:= new Process;
        Process_5:= new Process;
        Process_6:= new Process;
        Process_1.Initialize (P1); – Give each process its id.
        Process_2.Initialize (P2);
        Process_3.Initialize (P3);
        Process_4.Initialize (P4);
        Process_5.Initialize (P5);
        Process_6.Initialize (P6);
        Count1. Initialize (2, Process_3); – Initialize a join task
        Process_1.Fork;
        Process_2.Fork;
        Count2. Initialize (3, Process_6);
        Process_4.Fork;
        Process_5.Fork;
end Fork_Join;
```

Using task types and pointers (access types), a relatively simple program is written. Use of the case statement within the Process task type further simplifies the program This statement allows the use of a task type (instead of six tasks) to implement the concurrent statements. Each process is accessed through the use of its variable (pointer variable) and its identification determines which statements it executes. This model can be given to students and various exercises assigned which illustrate the power, as well as the difficulty, of using the ***Fork/Join*** constructs.

Semaphores

Semaphores are a well known general purpose construct used to solve synchronization problems. Traditional semaphores are nonnegative integer variables **V** (s) and **P** (s). We can define these operations as follows:[6]

- **V (s): Increment s by 1 in a single indivisible action.**
- **P (s): Decrement s by 1, if possible. If s = 0, then it is not possible to decrement s and still remain in the domain of nonnegative integers, the process invoking the P operation then waits until it is possible. The successful testing and decrementing of s are also an indivisible operation.**

FIGURE 5. Semaphore Implementation: Code and Process Graph

(a + b) * (c + d) - (e / f)

t1:= a + b

t2:= c +d

t3:= e / f

t4:= t1 * t2

t5:= t4 - t3

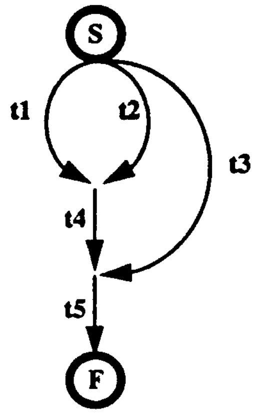


FIGURE 6. Semaphore Implementation: Ada Code [8]

```
with Text_IO; use Text_IO;
procedure Equation is

     -- PROGRAMMER: James L. Irvin

     A: constant Integer:= 2;
     B: constant Integer:= 3;
     C: constant Integer:= 4;
     D: constant Integer:= 1;
     E: constant Integer:= 2;
     F: constant Integer:= 2;

     package Int_IO is new Integer_IO (Integer);
     use Int_IO;

     T1, T2, T3, T4, T5: Integer;

     task One; task Two;
     task Three; task Four;
     task Five;

     task type Semaphore is
          entry Signal;
          entry Wait;
     end Semaphore;

     Sem1, Sem2, Sem3, Sem4: Semaphore;

     task body Semaphore is
          begin -- Semaphore
               loop
                    select
                         accept Signal;
                         accept Wait;
                    or
                         terminate;
                    end select;
               end loop;
          end Semaphore;

     task body One is
          begin -- One
               T1:= A + B;
               Sem1.Signal;
          end One;

     task body Two is
          begin -- Two
               T2:= C + D;
               Sem2.Signal;
          end Two;

     task body Three is
          begin -- Three
               T3:= E / F;
               Sem3.Signal;
          end Three;

     task body Four is
          begin -- Four
               Sem1.Wait;
               Sem2.Wait;
               T4:= T1 * T2;
               Sem4.Signal;
          end Four;

     task body Five is
          begin -- Five
               Sem3.Wait;
               Sem4.Wait;
               T5:= T4 - T3;
               Put (T5);
          end Five;

begin -- Equation
     null;
end Equation;
```

An Ada simulation need not concern itself with the counter *s* because any invoking process waits (if necessary) on a P or V operation when attempting a rendezvous with the semaphore. This implementation is effective, simple, and models semaphore semantics quite well. The implementation uses signal and wait for P (s) and V (s), respectively. In this exercise students were required to decide how many semaphores were necessary and implement the process graph depicted in figure 5. More difficult exercises can be assigned as required.

Monitors

A monitor is another well known mechanism for synchronization control. The simplest form of a monitor restricts access to the data structure through a set of operations and use of the monitor to one process at a time. This form of monitor is used quite extensively in concurrent Ada programs and is sufficient to provide a basic "feel" for their usage. True monitors [7] can also be implemented but these implementations (at least all of the methods we have attempted, so far) are quite convoluted and tend to hinder rather than aid the process of understanding.

Figure 7 contains an implementation of the basic monitor model. This monitor controls all access to a game board which contains ***n*** warship tasks and ***m*** mines. The warships can fire rounds, move to another location, be hit by unfriendly fire, and sink. To do any of these things a warship must make a request and/or report to the game monitor. The monitor effectively maintains the integrity of the resource (the game board).

Path Expressions

Path Expressions "provide a mechanism with which a programmer specifies, in *one* place in each module, all constraints on the execution of operations defined by that module".[2] Additionally the compiler generates code to enforce these constraints. Path expressions can be defined as follows:[6]

A path expression has the form: path *restriction_expression* end where *restriction_expression is defined recursively as follows:*

1. A procedure name *P* is a *restriction_expression; by itself, a single procedure name implies no restriction.*
2. If *P1* and *P2* are *restriction_expressions*, then each of the following is also a *restriction_expression*:

   P1, P2 denotes *concurrent execution*. No restriction is imposed on the order in which *P1* and *P2* are invoked or in the number of concurrent invocations.

   P1; P2 denotes *sequential execution*. One invocation of *P1* must complete before each invocation of *P2*. The execution of *P2* in no way inhibits the initiation of *P1*; thus many different invocation of *P1* and *P2* may be active concurrently, as long as the number of *P2*'s that have begun execution is less than the number of *P1*'s that have completed.

   n:(P1) denotes *resource derestriction*. It allows at most *n* separate invocations of *P1* to coexist simultaneously.

   [P1] denotes resource derestriction. It allows an arbitrary number of invocations of *P1* to coexist simultaneously.

The operators can be combined to express flexible and powerful constraints in concise notation. Implementation of path expressions in Ada can be problematic. The ideal situation would be to have a path expression parser. The parser would decipher the path expression and schedule the tasks as necessary. Alternatively a path expression can be given to students for conversion to Ada. The problem with the latter approach is that the constraints are no longer automatically enforced. Figure 8c indicates that more complicated constraints can be converted using a manager task to ensure that the constraints are met. If a certain pattern is to be reused often, it can be encapsulated in a package.

FIGURE 7. Monitor Implementation [9]

```
-- Programmer: Sherman White
-- This is the Game Monitor:
-- It is responsible for coordinating game play activity. It provides
-- all access to the Game Board through Shoot, Move, Initialize, and
-- status facilities.
task Game_Monitor is
        -- Initialize gives the Ship (This_ShipID) two legal coordinates on the
        -- game board.
        --
        entry Initialize (This_ShipID: in Ship_ID_Type;
                                        Row, Column: out Integer);
        -- Move moves a ship from its current position to a new position, if the
        -- ship is blocked by another ship, the advancing ship pauses.
        --
        entry Move (This_ShipID: in Ship_ID_Type;
                                FromRow, FromColumn,
                                ToRow, ToColumn: in integer;
                                ValidRow, ValidColumn: out integer);
        -- Shoot fires a shot at the given row and column in the game board. Shots
        -- are clustertized in that in addition to the target row and column, blocks
        -- adjacent to the target area are also "hit".
        --
        entry Shoot (ShipID: in Ship_ID_Type;
                        Row, Column: in Integer);
        -- Status returns the current status of This_ShipID; either Ok, Dead, or Win
        --
        entry Status (This_ShipID: in Ship_ID_Type;
                                Row, Column: in Integer;
                                RetStatus: out Status_Type);
end Game_Monitor;

task body Game_Monitor is
        type Board_Type is
                array (0..MAXIMUM_ROWS, 0..MAXIMUM_COLS) of Ship_ID_Type;
        Game_Board: Board_Type; -- the play areas
        ShipHits: array (Ship1..Ship6) of Integer:= (0, 0, 0, 0, 0, 0);
        R, C, Ri, Cj, Mine_Row, Mine_Column, DeadShips: Integer;
        --
        -- The Wrap Function changes the Game Board into a Torus and prevents
        -- constraint errors. Ships leaving the boundaries of the board are
        -- restored to positions on the other side of the game board.
        --
        function Wrap (i: integer; Limit: integer) return integer is
        --
        -- i <- Horizontal or Vertical Position
        -- Limit <- Maximum value which /i/ must never reach
        -- ** Note: The Minimum value for /i/ is assumed to be 0.
        --
        Result: integer;

        begin
                if i >= Limit then -- check upper boundary
                        Result:= i - Limit; -- reorient on lower side of game board
                elsif i < 0 then -- check lower boundary
                        Result:= Limit + i; -- reorient on upper side of game board
                else
                        Result:= i; -- if legal value, modify nothing
                end if;
                return Result;
        end Wrap;

        begin -- Game Monitor
                -- Place 10 Mines in the Play Area:
                --
                for i in 1..10 loop
                        loop -- Make Sure we get a legal position (not overwritten)
                                Random (Mine_Row, MAXIMUM_ROWS);
                                Random (Mine_Column, MAXIMUM_COLS);
                                if Game_Board (Mine_Row, Mine_Column) = NoShip then
                                        exit; -- leave when we find legal coordinates
                                end if;
                        end loop;
                        -- Place the mines
                        Game_Board (Mine_Row, Mine_Column):= ShipK;
                end loop;

                -- Game Event Loop:
                loop
                        select
                                accept Initialize (This_ShipID: in Ship_ID_Type;
                                                                Row, Column: out Integer) do
                                        loop -- find legal coordinates to place the ship in
                                                Random (R, MAXIMUM_ROWS);
                                                Random (C, MAXIMUM_COLS);
                                                 if Game_Board (R, C) = NoShip then
                                                        exit; -- leave loop when legal coordinates found
                                                end if;
                                        end loop;
                                                Game_Board (R, C):= This_ShipID;
                                                Row:= R; Column:= C;
                                end Initialize;
                        or
                                accept Move (This_ShipID: in Ship_ID_Type;
                                                FromRow, FromColumn,
                                                ToRow, ToColumn: in integer;
                                                ValidRow, ValidColumn: out integer) do
                                         if (Game_Board (ToRow, ToColumn) = NoShip) or
                                                (Game_Board (ToRow, ToColumn) = ShipK) then
                                                Game_Board (FromRow, FromColumn):= NoShip;
                                                -- If no opposing ship in path, move to new position
                                                ValidRow:= ToRow; -- Validate New coordinates
                                                ValidColumn:= ToColumn;
                                                if Game_Board (ToRow, ToColumn) = ShipK then
                                                        ShipHits (This_ShipID):= 10;
                                                else
                                                        Game_Board (ToRow, ToColumn):= This_ShipID;
                                                end if;
                                        else -- opposing ship was found
                                                ValidRow:= FromRow; -- pause at current position
                                                ValidColumn:= FromColumn;
                                        end if;
                                end Move;
                        or
                                accept Shoot (ShipID: in Ship_ID_Type; Row, Column: in Integer) do
                                        for i in 1.. 3 loop -- use cluster bomb effect
                                                for j in 1.. 3 loop
                                                        Ri:= Wrap ((i-2) + Row, MAXIMUM_LIMIT);
                                                        Cj:= Wrap ((j-2) + Column, MAXIMUM_LIMIT);
                                                        if (Game_Board (Ri, Cj) /= NoShip) and
                                                                (Game_Board (Ri, Cj) /= ShipK) then
                                                                ShipHits (Game_Board (Ri, Cj)):=
                                                                ShipHits (Game_Board (Ri, Cj)) + 1;
                                                                if ShipHits (Game_Board (Ri, Cj)) >=
                                                                        MAXIMUM_HITS then
                                                                        Game_Board (Ri, Cj):= NoShip;
                                                                end if;
                                                        end if;
                                                end loop;
                                        end loop;
                                end Shoot;
                        or
                                accept Status (This_ShipID: in Ship_ID_Type;
                                                Row, Column: in Integer; RetStatus: out Status_Type) do
                                        if ShipHits (This_ShipID) >= MAXIMUM_HITS then
                                                RetStatus:= Dead;
                                        else -- check for win status
                                                DeadShips:= 0;
                                                for i in Ship1..Ship6 loop
                                                        if i /= This_ShipID then
                                                                if ShipHits(i) >= MAXIMUM_HITS then
                                                                        DeadShips:= DeadShips + 1;
                                                                end if;
                                                        end if;
                                                end loop;
                                                if (ShipHits(This_ShipID) < MAXIMUM_HITS) and
                                                        (DeadShips = MAXIMUM_SHIPS-1) then
                                                        RetStatus:= Win;
                                                else -- if we haven't won and we aren't dead, we're Ok
                                                        RetStatus:= Ok;
                                                end if;
                                        end if;
                                end Status;
                        or
                                terminate; -- This means that the other processes terminated.
                        end select;
                end loop;
        end Game_Monitor;
```

FIGURE 8. Path Expression Templates

```
-- path 1:(T1), (T2) end
-- Maximum of one T1 and T2
--can operate concurrently.
procedure Concurrent is
      task T1;
      task T2;

      T1 body is
      .
      .
      end T1;

      T2 body is
      .
      .
      end T2;

begin -- Concurrent
      null;
end Concurrent;
```

FIGURE 8a

```
-- path T1; T2 end
procedure Sequential is
      task T1 is
            entry Startup;
      end T1;

      task T2 is
            entry Startup;
      end T2;

      task body T1 is
            begin -- T1
                  loop
                        select
                              accept Startup;
                        or
                              terminate;
                  end loop;
                  .
                  .
            end T1;task body T1 is
            begin -- T1^2
                  loop
                        select
                              accept Startup;
                        or
                              terminate;
                  end loop;
                  .
                  .
            end T2;

begin -- Sequential
      For Counter in 1..10 loop
            t1.Startup;
            t2.Startup;
      end loop;
end Sequential;
```

FIGURE 8b

```
-- path Counter1: (Counter2: (T1), Counter3: (T2) end
procedure More_Complicated is
      task T1 is
            entry Startup;
      end T1;

      task T2 is
            entry Startup;
      end T2;

      task Manager is
      -- Manager is responsible for controlling access to
      -- T1 and T2. A maximum of Counter2 invocations
      -- of T1 and Counter3 invocations of T2 can
      -- proceed concurre.  y as long as no more than
      -- Counter1 total invocations is not exceeded.
            entry Fire_T1;
            entry Fire_T2;
      end Manager.

      -- Optional other tasks

begin More_Complicated
-- Invocations of T1 and T2 are made through the
-- manager. They can be made from here or other
-- optional tasks.
end More_Complicated;
```

FIGURE 8c

Ada for programming exercises

A wide range of programming exercises can be assigned. The difference in programming exercises and expository assignments is that programs are **completely specified** real-world problems which are amenable to concurrent solutions. The student is required to design and implement his solution from the specification and is not restricted to one model or notation. Simulations and real-time problems are but a few possibilities. Students gain invaluable experience in concurrent program design and development.

Roller Coaster [10] (Simulation)

Suppose there are *n* passenger and *m* ($m > 1$) car process. The passengers repeatedly wait to take rides in the cars, which can hold *C* passengers, $C < n$. However, the cars can go around the tracks only when they are full. Since there is only one track, cars cannot pass each other; i.e., they must finish going around the track in the order in which they started. Write a program which implements this. Use message passing for communication. Assume there will be 1058 passengers. Use five cars.

This is an excellent middle level program. It requires the various tasks to interact in a non-trivial manner, yet it will not overwhelm the student. A typical (if sparse) design may be as follows.

FIGURE 9. Roller Coaster Design

Generate_Passengers

```
for Index in Pax_Range loop
    Roller_Coaster.Get_Pax;
    Random_Delay;
end loop;
Roller_Coaster.No_Passengers;
```

Car_Task

```
loop
    select
        accept Get_ID (ID: Integer) do
            My_Id:= ID;
        end Get_ID;
    or
        accept Start_up;
        Delay (Trip_Time);
        if not Roller_Coaster.Terminated
        then
            Roller_Coaster. Done (My_Id);
        end if;
    or
        terminate;
end loop;
```

Roller_Coaster

```
Make queue of Max_Cars, giving each car a unique ID;
Count:= 0;
loop
    loop
        select
            accept Get_Pax;
            Count:= Count + 1;
            if Count = Max_Pax then
                Dequeue (A_Car, Queue);
                A_Car.Start_Up; Count:= 0;
            end if;
        or
            accept Done;
            Enqueue (Done_Car, Queue);
        or
            accept No_Passengers;
            while Cars are still out loop;
                accept Done;
            end loop;
            Shut_Down:= true; exit;
        end select;
    end loop;
    If Shut_Down then exit
end loop;
```

Although this design may seem to be simplistic, students typically have some difficulty arriving at this stage. For most of them it is their first time actually designing and implementing a concurrent program and defining the interfaces is not trivial. However, once they work through early mistakes, most of them produce very good programs. Most problems occur in visualizing the necessary interactions between the objects. Students are so used to sequential programming that they have a difficult time realizing that rendezvous' are not procedure calls. The various tasks are really executing concurrently. The earlier models/primitives mask this somewhat, because of the low-level coding required to implement concurrency in a program (sem1.wait; sem2.signal;......; sem1.signal; sem2.wait). Using message passing eliminates the need for this low level coding, but requires thinking about concurrent programs differently, especially when the responsibility for program design, as well as implementation, is the student's responsibility.

Mongulator (Real-Time Programming)

Mongulator (a nonsense name) was originally conceived as a simple real time program. It turned out to be quite difficult for the students. The original specifications (which follow) were not adequate and we had to discuss them quite thoroughly throughout the development process. They also required subsequent modification. It was deemed acceptable to accept some trashing, but each assembly line was still required to contain the intelligence to determine its speed.

1. A factory has 4 assembly lines. Each line produces a part required to produce the Mongulator. The following list indicates the maximum speed in which the respective lines can produce its component.

   Line A: 0.5 seconds

   Line B: 0.25 seconds

Line C: 0.33 seconds

Line D: 0.47 seconds

2. Each line delvers its part to a collector. The line can not wait more than 0.05 seconds for a delivery to be made. If a delivery can not be made then it is trashed. Records must be maintained of the number of lost parts.
3. The collector accepts the components is some order every 0.5 seconds and forwards a complete colle‿tion to the assembler. The collector only waits 0.005 seconds to collect a component from each assembly line. The collector has the luxury of buffering two incomplete collections.
4. The assembler assembles a collection in 0.5 seconds. It can also buffer one collection while assembling another.
5. Write a program which minimizes or eliminates trashing.

   In a real factory the speed of the assembly lines varies. Suppose each line starts at its maximum speed. If it senses that it is going too fast, then it slows itself. If it senses that it is going too slow, it attempts to speed up (its maximum value can not be exceeded.

   We should never trash.....

What this program reaffirmed to both the instructor and students is that real-time programming is at least one degree of difficulty above "ordinary" concurrent programming. Careful design and a thorough understanding of the specifications is mandatory. Extreme care must be taken to implement the design with accuracy. Testing of the program must also be thorough and good fault isolation skills are required to correct problems.

After completing this or a similar project, students have the basic tools and understanding to begin more serious real-time projects.

FIGURE 10. Mongulator.Ada program fragments

```
task Opening_For_The_Day;
task Collector;
task Assembler is
    entry Assemble_Part;
end Assembler;
task type Line_Type is
    entry Initialize (Id: in character;
        Time: in duration;
        Percent_Error: in Float);
    entry Part_Received;
    entry Update;
    entry Closing_Time
        (Percent: out Float;
        Line_Time: out Duration;
        Pause_Time: out Duration);
end Line_Type;

Line: array ('a'..'d') of Line_Type;
```

Task Opening_For_The_Day

```
begin -- Line_Type
    accept Initialize (Id: in Character;
            Time: in Duration;
            Percent_Error: in Float) do
            Line_Id:= Id; Speed:= Time;
        Trash_Margin:= Percent_Error;
    end Initialize;
    loop
        delay (Speed);-- part is being made
        Made:= Made + 1;
        select
            accept Part_Received;
        or
            delay (Pause); Trash:= Trash + 1;
        or
            accept Update;
            Prcnt:= 100.00 * (Float(Trash)
                    / Float(Made));
            if Prcnt > Trash_Margin then
                Pause:= Pause + Time_Increment;
            end if;
            Total_Made:= Total_Made + Made;
            Total_Trash:= Total_Trash + Trash;
            Made:= 0; Trash:= 0;
        or
            accept Closing_Time (Percent: out Float;
                Line_Time: out Duration;
                Pause_Time: out Duration) do
            Percent:= 100.00 *
            Float(Total_Trash) / Float(Total_Made);
            Line_Time:= Speed; Pause_Time:= Pause;
            end Closing_Time;
        end select;
    end loop; end Line_Type;
```

Task Type Line_Type

```
begin -- Collector
    loop
        for Count in 'a'..'d' loop
            select
                Line(Count).Part_Received;
                if Bin(1,Count) = Empty then
                    Bin(1, Count):= Count;
                elsif Bin(2, Count) = Empty then
                    Bin(2, Count):= Count;
                end if;
            or
                -- waiting for the line to deliver a part
                delay (Collecting_Delay);
            end select;
        end loop;
        Send(1):= True;
        Send(2):= True;
        For X in 1..2 loop
            For Y in 'a'..'d' loop
                -- Check to see if bin full
                if Bin (X, Y) = Empty then
                    Send(X):= False;
                end if;
            end loop;
            -- If full bin send to assembler
            if Send(X) then
                Assembler.Assemble_Part;
                For Z in 'a'..'d' loop
                    Bin(X,Z):= Empty;
                end loop;
            end if;
        end loop;
    end loop;
end Collector;
```

Task Collector

Due to the complicated nature of the program only task specifications and fragments for two tasks are shown in figure 10.

The tasks conform to the following descriptions:

1. **Task Opening_For_The_Day**

   Opening_For_The_Day gets the percent of trash error for the day and passes it each line along with the line's identification and production speed.

2. Task Type Line_Type

   Line_Type is a task type that represents an assembly line in the "factory" A task of this type will make a part at the speed given to it by the Opening_For_The_Day task. It will then pause and wait for the collector to pick up the part on a default interval of 0.05 seconds. If the part is not collected it is added to the trash. When the assembler signals, the line will compute its trash ratio and if it is higher than the given trash margin it will attempt to slow itself by adding to the amount of time it will wait for the collector to collect a part.

3. Task Collector

   Collector collects the parts from each line. It will place the part in one of two bins unless the part is already in the bin. If the part is in the bin then the collector "lets the part fall on the floor." If one of the bins is full then the collector "sends" it to the assembler. If the collector finds a line is not ready to deliver its component, it will wait 0.05 seconds before moving on to the next line.

4. Task Assembler

   Assembler "assembles" the parts "collected" from the Collector task at a rate of 0.5. It will also prompt the lines to update every time it assembles 20 parts, and when it has assembled 10,000 parts it will prompt the lines to report its final results and quit for the day. The assembler will also prompt the collector to quit when it has assembled 10,000 parts.

The program fragments and task descriptions indicate an OOD approach to solving the problem. Each task or task type is treated as an object. The external interfaces were decided and the internal processing of each object were then designed and implemented. Ada's modular tasking mechanism decreased the complexity of the problem significantly.

Nothing is Perfect

Simulation

While Ada's concurrency mechanisms allow us to "simulate" many of the primitive synchronization constructs and concurrency models, it must be emphasized to the students that these simulations are not perfect and are primarily used to allow them to become familiar with the advantages and disadvantages of each construct and model. When used in this manner, simulation can have a big pay off in the classroom.

Information Hiding

One problem, which needs to be addressed, occurs when the implementation details of the simulation are not encapsulated in packages. If the details are incorporated directly in the their programs (as in all of the examples in this paper), some students will assume that semaphores (or some other structure) are naturally built using rendezvous' and require tasks with entry calls, etc. Therefore, instructors must be careful to distinguish between the implementation of the simulation and the use of the construct. While it is more awkward (and less instructive) to provide the constructs through Ada packages (thus hiding implementation), some confusion may be alleviated.

Parametized Initialization

Not having the ability to initialize Ada tasks through parameters is an irritant. Additional entries have to be included in each task to acquire identification and start-up information.

Troublesome Select Statement

Students must be cognizant of problems which can occur if tasks are symmetrical [11] and conditional entry calls are utilized. More generally, most of the objections discussed by Gehani and Cargill are pitfalls which students should know about.

Conclusions

Ada is an excellent language to use as an expository tool. It can be used to allow students to effectively write programs using the programming notations and synchronization primitives normally taught in beginning Operating Systems and Concurrency classes. It is an invaluable tool, in that, students can concentrate on learning the advantages and disadvantages of each construct through practical experience, as well as classroom discourse. Instructors can provide students with example implementations and tailor exercises to fit the needs of the class.

References

[1] Gehani, N. and McGettrick, A., **Concurrent Programming**, Addison-Wesley, 1988.

[2] Andrews, G.R. and Schneider, F., **Concepts and Notations for Concurrent Programming**, ACM Computing Surveys, 1983.

[3] Willis, R. and Morell, L., **Intelligent Abstract Data Types and Beyond**, to appear.

[4] Sebesta, R., **Concepts of Programming Languages**, Benjamin Cummings, 1989.

[5] Peterson, J. and Silbershatz, A., **Operating System Concepts**, Addison Wesley, 1985.

[6] Bic, L. and Shaw, A., **The Logical Design of Operating Systems**, Prentice Hall, 1988.

[7] Hoare, C. A. R., **Monitors: An Operating System Structuring Concept**, Communications of the ACM, 17 (10), 549-557.

[8] Irvin, J., **Semaphores.Ada**, Hampton University CSC 395, Spring 1991.

[9] White, S., **War_Ship.ada**, Hampton University CSC 395, Spring 1991.

[10] Andrews, G. R., **Concurrent Programming: Principles and Practice**, Benjamin/Cummings, 1991

[11] Gehani, N. H., Cargill, T. A., **Concurrent Programming in the Ada Language: The Polling Bias**, Software: Practice and Experience, 14 (5), 413-427, John Wiley and Sons, Ltd.

Lessons Learned in the Ada Training Programs at Rockwell

Mary Kathleen (Kt) Cook
William D. Baumert

July 1991

Collins Commercial Avionics
Rockwell International
Cedar Rapids, Iowa

INTRODUCTION. Collins Commercial Avionics, and Collins Avionics and Communications Division, Rockwell International, in Cedar Rapids, Iowa, have long recognized the need to provide continuing education opportunities for their employees. Software is a prominent part of the company's products. Thus, in order to maintain a competitive edge and to prevent erosion of technical competence, an extensive educational program has been determined to be imperative. In order to focus on the educational needs of Rockwell employees, there are several such training programs underway. This paper focuses on Ada training in Cedar Rapids, and the "lessons learned" along the way.

OVERVIEW

BACKGROUND. Both software engineering and Ada training have been heavily emphasized since before 1987. In 1987, a formal Software Engineering Training Program (SETP) was established to address the need for a more complete software engineering education program for Collins engineers. For FY91, this program features around 30 courses offered on a continuing basis, with other "short courses" offered on a more limited schedule.

One portion of the SETP courses is the Ada training block. The Ada courses were introduced specifically to begin to meet the demand to utilize Ada in accordance with Department of Defense requirements. Ada was also chosen to be the vehicle for illustration of the software engineering principles and goals. These have included such courses as Ada for Managers, Principles of Design, Ada Specific Design Issues, Ada Coding Issues, and Advanced Ada, in addition to several short courses. The number of hours required to complete one of these courses ranges from 4 to 80. More than 700 participants have been involved with the Ada training program as of September 1990.

FACILITIES. Education facilities were built to accomodate these courses. An education building housing several training rooms was established. One room is equipped with terminals and ports allowing each participant to have access to a Digital Equipment Corporation VAX cluster and specifically to Ada compilers. Another room has been equipped with video equipment so that many of the courses may be taped while being offered live. Another classroom is furnished with video equipment to facilitate courses obtained through satellite down-links, or for the self-paced viewing of pre-recorded lectures. Additional training rooms are available in this building as well as throughout the Rockwell complex. Off-site facilities have also been used on occasion.

TRAINING ASPECTS. The training program itself is extremely dynamic. The nature of training demands that we always seek to improve existing materials, and to add, modify, or delete courses as our needs and technology changes. There are two key factors to maintaining a quality training program. First, it is necessary to identify areas of success and to continue in them. Secondly, the identification of problem or challenge areas, and finding resolutions is vital. The management of these challenge areas not only keeps the classes interesting and current, but effective as well.

AREAS OF SUCCESS

It is interesting that each of our areas of success has also been a challenge area in many ways, and it is difficult to separate the two categories.

WORKSHOPS. One activity that has proven to be invaluable to our Ada courses is the inclusion of group workshops or projects. These exercises are sprinkled throughout the courses and between every major topic.

It is no surprise that the majority of learning actually takes place in these workshops and not in the standard lecture time. In the design and coding classes at least 50% of the students' time is spent on group projects. The first couple of workshops are very brief, taking only about 30 minutes. As the courses progress, the exercises become more substantial, taking up to two or three class periods to prepare and present. Our group projects offer several positive experiences in addition to simply reinforcing the material presented in lectures.

There are three phases of each Ada design workshop exercise. First, groups are chosen and then meet together and design a specified system. Then several of the groups present their designs in a more formal setting before the whole class. Following these presentations, everyone participates in a thorough design review.

It is extremely helpful for the members to learn to work in-the-large with all other members of the class in groups that change from one workshop to the next. Furthermore, the whole experience of working in teams seems to add to the classroom environment of cooperation, and sometimes competition, making the course far more interesting and more beneficial overall, than standard lectures. Over and over again, we have seen this team spirit motivate groups to go to great lengths to produce amazingly complete, professional looking designs for class presentations. Data Flow Diagrams and other materials from SA/SD, Control Flow Diagrams, Object Oriented Designs, and other areas beyond the scope of the class have made their way into group designs. This not only demonstrates the level of group enthusiasm that these teams seem to generate, but also attests to the influence of the entire Software Engineering Training Program on the general level of skills and awareness throughout the company.

All class members are strongly encouraged (or else are 'chosen to volunteer') to present workshop produced designs in front of the larger group. This practice helps to turn a rather intimidating activity into a routine skill with much less personal sensitivity and pressure. It is most encouraging to see all members of the class become able to make a formal design presentation in a fairly non-threatening environment.

The emphasis on reviews brings the goals and principles of software engineering together with direct practical applications for improving designs and the design process. As these workshop reviews evolve, the level of communication and constructive criticism goes up dramatically. In turn, the quality of the completed designs also improves dramatically. Even the quality of the presentation itself often goes well beyond the workshop requirements, involving extensive use of outside tools, formal graphical representations and laser printed design packets. The final workshop for the Design With Ada class has evolved into a very elaborate project not because we have changed the requirements, but primarily because the students themselves have produced increasingly complete designs in order to pass the scrutiny of peer reviews. For this reason, it was recently factored out into a course of its own to allow for the extra time and to encourage that level of effort.

TEAM-TEACHING. The team teaching approach has contributed to the success of the Ada training program. By having two or more instructors teaching each of the major Ada courses we are able to divide the materials and specialize on certain topics. There is always someone else in the room able to verify or look up answers to student questions that may not otherwise be possible during lecture time. Having backup instructors prevents us from ever having to cancel a class due to unforseen instructor absences. We have had a nice mix of experience and training/education between the various instructors who come from different departments, divisions, and projects throughout Rockwell. This has enhanced our collective credibility in many different ways.

An important facet of team teaching at Rockwell involves treating the class as a total concept and not just a set of disjoint ideas, days, chapters, concepts, etc. Instructors do not simply show up for 'their chapters' or 'their days' and teach. There has to be a continuity and consistency between all class concepts. Everything must fit into a total picture for the class to make sense. For this reason all instructors are capable of teaching any unit of a given class, and new instructors will usually sit through at least two or three offerings before they begin to take on a lead instructor role.

To some extent, this emphasis on continuity even overlaps between different classes. The instructor must be most familiar with the concepts taught in the prerequisite courses, and how topics will be dealt with in subsequent classes. References are often made to exercises and concepts covered in prerequisite courses. Again, everything must fit into a consistent total picture.

CHALLENGE AREAS

We continue to address challenging issues in the training efforts. These issues can be roughly grouped into two main categories: General Education Issues, and Software Engineering and Ada Industry Specific Issues.

GENERAL EDUCATION CHALLENGES

First, within the General Education area, we continually address the various difficulties that are inherent to any kind of training in general.

- Management Support
- Motivation
- Classroom atmosphere
- Variance of participant's backgrounds
- Cost and sources of training
- Timing and scheduling
- Staffing

Although some of these may seem like secondary concerns in the Ada arena, they are nonetheless very real, and contribute vastly to the success or failure of any training program, in any discipline.

MANAGEMENT SUPPORT. The past (and often present) attitude toward industry training in general has been that although everyone agrees on the value of trained people, it is not "real work" and has been paid little more than lip service when it comes to investing and committing skilled designers and engineers to a lengthy training program. As B. Boehm stated in Software Engineering Economics, "training and human relations activities provide by far the largest source of opportunity for improving software development and productivity." And as K. Blanchard and S. Johnson pointed out in The One Minute Manager, "It's ironic... Most companies spend 50% to 70% of their money on people's salaries. And yet they spend less than 1% of their budget to train their people. Most companies, in fact, spend more time and money on maintaining their buildings and equipment than they do on maintaining and developing people."

Rockwell has demonstrated its committment to training. Managers, of course, are equally frustrated with the state of software development, and are seeking for understanding and solutions to the dilemma. Over the past several years we have definitely progressed from the "What are we going to do about this 'Ada thing'" stage, to a high level of management awareness, and more importantly, involvement.. We have hardware organizations, software organizations, and systems organizations. Members of these organizations are now assigned to all sized projects. There are more and more managers at Rockwell that have come from a predominately software background, which also represents progress in a traditionally hardware intensive company. We have offered a class entitled Ada for Technical Managers that addressed the specifics of both software engineering and Ada that assisted managers in supporting the engineers doing Ada development through a better understanding of the concepts that they have been learning in the Ada program. This class served as a small part of the progression towards better management support and motivation. Comments made in the Ada for Technical Managers class indicated that there is a growing realization that the principles of software engineering pay off.

MOTIVATION. Another challenge for any type of training program is correctly motivating the students. In a University setting, classes are taken for credits, fulfillment of requirements, and to get training in areas that will eventually lead to employment. Thus, even when the topic is not necessarily the most interesting, generally students will hang on. Here the motivations are different. Most everyone agrees that additional training will enhance an individual's performance and growth in job skills, but the immediate return for taking time and effort from pressing job responsibilities is not as apparent. It can be difficult to motivate participants towards new or difficult concepts if they are not convinced that they will be of any immediate value to them. Our courses are not generally required. They are strictly elected by the participants and sometimes by their managers. Essentially, we have to be able to answer the unspoken questions like, "Is this worth my time", and "Am I learning, understanding and enjoying the class, or am I just sitting here wishing that I could go home early?"

Generally, motivation begins long before the class starts, with the enthusiasm and support of individual managers, as discussed. However, we must continue to motivate the students in the classroom. It is not at all optional that the class be interesting and relevant. Were this not the case, the class would quickly die in the wake of severe criticism and disinterest. Of course one must take a critical look at such sizeable investments of time, and through avenues like class reviews, and the SETP Planning Committee, inadequate classes are readily identified and changed or omitted.

The techniques of motivating students in the Ada classrooms are no different than motivating any type of student. The instructors must utilize good presentation skills, and never waste class time. New or difficult topics should be presented carefully and slowly without allowing distractions. Continuous monitoring of class understanding and interest is achieved through a fairly interactive teaching style. It can be difficult to control the flow if students are allowed to drift off onto tangents by answering, and thereby encouraging too many questions that are of the "what if" variety. The "what if" game becomes contagious and can lead the discussion far from the intended objectives. These tend to cause students to miss the forest for the trees. However, this has to be handled very carefully. In our business, it is exactly these types of questions that engineers like to ask, are good at asking, and ultimately *have* to ask if they are to succeed. To simply dismiss them would cause hostility and an immediate loss of interest in the class. If handled with respect and clear explanations of the intent, most participants actually appreciate the direction of the course and the attempt to keep the time spent as worthwhile as possible. Here again, it helps to know exactly what will be covered in subsequent courses, in order to know what level of detail is appropriate for responses.

It is also important how instructors respond to students and their needs during non-class hours. Exhibiting a willingness to accept all class members as people with technical skills, ideas, experiences, and needs, both personal and technical, is important. We definitely assume that all class participants are intelligent, but are careful NOT to assume that they are always getting the point or concept. Just as the instructors are provided with an expanding knowledge base through opportunities to attend other classes, conferences, symposiums, the experience and backgrounds of attendees can augment class discussions if encouraged and monitored.

CLASS ATMOSPHERE. In keeping with the need to motivate students, the classroom atmosphere is a key factor and must be pleasant, not formal or dull. To sit through a class for four to eight hours per day is tedious enough to begin with. The frequently scheduled workshops, breaks, topical anecdotes, and the flexibility to change the schedule as needed keep a long day from becoming interminable. Special "side shows" are kept on hand for long, or tiring stretches. One of these "side shows" that has been fun, and certainly served its purpose is to turn various exercises into small competitions. One exercise in the Design with Ada class, that reviews the definitions of various software engineering terms and expressions has been a good example of this. Instead of simply going through a fill-in-the-blank style questionnaire, these definitions have been laid out on an overhead to look like the gameboard for "Jeopardy". The class is divided into two groups, and groups are allowed to choose their "categories" such as "Levels of Cohesion", "Levels of Coupling", "Design Methodologies", and so on. Points are scored, and in the end, a 15 minute exercise has been far more interesting, and the material will be remembered much longer because of that. It has become almost a hobby for the instructors to collect these special cartoons, and activities, and to use them judiciously.

Group workshops also contribute to a positive classroom atmosphere. Groups are chosen in a continuously revolving fashion allowing the class to become well acquainted. The friendships, or level of comfort with others in the class, causes evaluations of presentations to be much less threatening and personal, and thus more honest and helpful. Also, this knitting of the "team", encourages self-motivation and creativity. Occasionally, it becomes clear who the independent workers are, or the more dominant group leaders, and then the instructors can select groups less randomly. Carefully monitoring progress during workshops allows for this "stacking" that also contributes to the success of workshops and less frustrations for others who might feel unable to challenge more forceful group members who tend to govern the whole project.

When dealing with the attitudes of attendees, we have attempted to understand the pressures they are currently facing, and to not be judgmental about their attitudes, willingness or unwillingness to accept what's taught, willingness to work on and cooperate during workshops, and occasionally, the need to let off a little steam in the way of fanatical opinions and soapboxes.

Just as we try to develop the ability to evaluate designs in an impersonal manner, the instructors too, must be able to accept criticism. All students have the opportunity to evaluate each course's materials and the instruction. Students criticism, suggestions, and comments help to further refine the program, and are always taken seriously. Also, it helps to be able to tell and take a joke. Finally, after laying an egg, it doesn't hurt to be able to stand back and admire it.

BACKGROUNDS. Another area of difficulty arises from the widely diverse backgrounds of participants in the class. It is seemingly impossible to be "all things to all people" and we continue to grapple with the level of detail that is presented in each course. Although a clear set of prerequisites and assumptions are defined to select who should attend each course, there are always circumstances that make them impractical. Among the more extreme examples, our courses have included participants that range from those who were on their first week of employment fresh out of college with a fairly good academic background in higher order languages and essentially no practical experience in real-time embedded systems, all the way to 25-year veteran hardware engineers with little or no language background outside of Assembly. In between these extremes are many engineers with software experience, possibly with Computer Science degrees, and a solid background in higher order languages as well as varied amounts of understanding of the goals and principles of software engineering.

Needless to say, putting individuals with that much diversity together in an Ada workshop setting has its challenges. We find merit in the endeavor simply by the fact that this is exactly how project teams sometimes are composed outside of the classroom setting. Just as the value of Concurrent Engineering has been demonstrated in the development environment, the practice of 'concurrent learning' has its place. An RF engineer with 15 years of experience certainly has a unique view of a system that more traditional software engineers can also benefit from. We take the perspective that there is something to learn from everyone, and that differing viewpoints that cause others to think about the problem in different ways are valuable. Each and every class seems to have its own "personality" and appropriate level for presentation. It is futile to take a dogmatic "this is how it's always been done before" approach, and the ability to tailor each class is essential to the overall success of the Ada training program.

PROCUREMENT. Considering the high cost of training, the decision between procuring training from vendors or development of courses in-house is not always an easy one. The SETP has become an assemblage of both. At present, the body of the Ada courses has been purchased from outside vendors. It has become apparent that some of the important aspects of these arrangements are the quality of the support, and the ability to establish ownership. Without having full ownership of the courses, Rockwell would be unable to tailor, change, or update courses, and they would quickly become obsolete. Presently, the courses are developed by outside vendors who provide all course materials that Rockwell then owns, and has the right to change at will. This ownership arrangement allows vendors to retain the right to sell and teach the courses elsewhere, while not unnecessarily restricting our uses. This has proven to be a useful set-up for all parties involved. These vendors generally will teach one or more iterations of the courses, making changes to the materials and to the content, as requested by Rockwell. Once these courses are suitable, then either the vendor continues to teach subsequent offerings, or, most often, Rockwell targeted instructors work closely with the vendor to assume responsibility for the classes. Because of this it is vital that vendors are able and willing to work closely with Rockwell instructors and to take an active role in the transition. It is not adequate for a vendor to provide course materials, teach one iteration, and then sublimely leave town. The level of support and "maintenance" of vendor supplied courses has been invaluable to our successfulness.

SCHEDULING. It is important to offer courses at convenient times and locations while balancing the amount of time that participants must spend away from their primary projects and responsibilities. All classes are scheduled on the Rockwell premises, although not in the primary office buildings. Classes must be scheduled to allow time for large amounts of material to be absorbed. However, going solid for too long, that is, long days of lectures across many consecutive days, doesn't work. Most projects can't afford to allow people to leave for long periods of time, and fatigue becomes a factor. Spreading things out too much disrupts the continuity. For these reasons, we have typically taught two iterations in a block fashion, 7-8 hours per day, every day until completion. This alleviates the backlog of individuals with urgent needs to have the material as fast as possible. Then we offer a more gradual schedule where classes will meet either 8 or 12 hours per week, distributed across two non-consecutive days, until completion. This type of schedule does not lend itself to classes taught by outside vendors, which is one several reasons why the major Ada classes are taught by Rockwell instructors.

STAFFING. As mentioned, it is sometimes difficult to fully separate the areas of success from the challenges. Because it is very time intensive for instructors to achieve the level of continuity discussed previously, it has not always been easy getting qualified instructors from within each of the Rockwell divisions. For those instructors who do invest the time and committment, often their efforts are not as visible to their managers and co–workers who are not in the classroom. Individuals with good Ada and software engineering skills are not only desireable as instructors, but are also highly in demand for project work. Therefore, there is a certain amount of attrition among trainers, and new trainers are always needed.

SOFTWARE ENGINEERING AND ADA INDUSTRY CHALLENGES

Training, as it relates particularly to the topics of software engineering and Ada, is not without its challenges. We will address the following:

* Applying the software engineering Principles and Goals

* Magnitude, Complexity, and Specifics of Ada
  — Pre–conceived prejudices to Ada

APPLICATION OF SOFTWARE ENGINEERING PRINCIPLES AND GOALS. In each of the Ada classes offered for the Avionics and Communications Divisions of Rockwell, in Cedar Rapids, there is a strong emphasis on the principles and goals of software engineering as defined by Ross, Goodenough, and Irvine in "Software Engineering: Process, Principles, and Goals", Computer, May 1975. We stress the trade–off between front–end costs and life–cycle costs. The need for a better understanding of a life–cycle is addressed. Realistic scheduling of time for projects to allow for complete analysis and design before pushing into implementation, has almost become a crusade. Emphasis is placed on the importance of computing various design metrics and then USING them to make improvements in the design before proceeding with implementation. We strongly encourage multiple walkthroughs and reviews at each phase of the life–cycle. Maintaining complete and current documentation at each level is also pushed. It is clear that not using these skills and steps from the onset of a project, is almost fatal by the maintenance phases. The cost and time of trying to produce documentation, and "fix" all the problems introduced by poor analysis and design is virtually unlimited.

The difficulty with teaching these concepts is not in the acceptance. Very few individuals would challenge the importance of each of the points. Rockwell is firmly committed to doing things right and producing quality products. The reality, and the problem across the entire software industry, is that applying these ideas is expensive, time–consuming and difficult to do with large project groups under great pressure to produce the finished product quickly. We realize and acknowledge that these pressures always exist, and attempt to teach and work towards the ideal.

As a result of this kind of focus in the Ada training program, students leave the classes with a better understanding of the life-cycle and an enthusiasm to make changes in the way that they produce software products. We feel one indication of this is that, out of better understanding, engineers are becoming more critical in several ways. There is a greater demand for support, and tools. With a better understanding of the various features and also a better understanding of our own requirements for these systems, much more care goes into the selection processes. We have become more critical about doing things right. In the Ada classroom we regularly experience a certain level of frustration about the state of Ada development and software engineering across the industry. The good news is that these feelings contribute towards change.

MAGNITUDE, COMPLEXITY AND SPECIFICS OF ADA. With regards to the language itself, there are some specific challenges. Although this is much less the case now, several years ago, we were forced to address many pre-conceived prejudices about Ada. Many of these ideas are no longer issues, some are still concerns, and some are basically true. Here is a sample of the statements and ideas that we have tackled.

Ada is too big, has too much to learn, and is too complex.
Ada I/O is awkward.
Code generation is poor because it's slow and inefficient.
Ada restricts the engineer's programming freedom (harder to hack).
It's not suitable for embedded systems or hard deadline scheduling
 because tasking is not deterministic. Hard to verify for FAA, etc.
Tools are immature and not supported.
Tasking is slow and cumbersome.
There is no object inheritance, so it is not really an object-oriented
 language like C++. How could it be better than C anyway?
Compilers are expensive, and may not exist for my machine.
Just when we get Ada right we'll have to cope with Ada9X, and who
 knows what that means.
It's hard to accept having a language mandated to me.

And finally, it is true that Ada has not saved software engineering and cured the ills of the software industry like people thought that it would. Ada has not eliminated or solved the software crisis. Ada is not, and never will be **The Solution** to all of our software woes, but many felt that the language was originally billed as just that. This alone has created a general feeling of disappointment and resistance to the language.

Ada is more than just syntax, and cannot be taught in one course, or by merely reading one book. If one takes all of the courses in the Ada training program in sequence, the introduction to the language comes as an off-shoot of the design class, and is used primarily as a representation mechanism for these designs. In fact, there are two complete courses offered as prerequisites to the Ada Coding classes.

TASKING. Specific features of the language are more difficult to teach and to understand than others. There has been a great deal of attention given to the Ada tasking mechanism. Many papers, and seminars have been devoted to this topic alone, so we'll not belabor that. Suffice it to say that we have found that when teaching Ada tasking, it is important to emphasize that it is indeed an elegant feature of the language, though there are realistic concerns. We address these concerns directly, explaining them and not skirting the issues. It is also emphasized that Ada compilers are improving, and tasking is becoming more efficient. Examples are used heavily, and we rely on workshops to help make clear the ideas of concurrency and the Ada-specific model.

PACKAGES. Introducing the topic of Ada packages and the packaging mechanism really seems to bring a focus to many of the things that cause Ada to rise above many other higher-order languages. This discussion gives us a chance to apply the goals and principles emphasized earlier by providing direct support for concepts such as abstraction, information hiding, and modularity. Adding the concept of private types shows how Ada gives users a way to actually enforce these important software engineering goals and principles. It is at this point that the students are really able to tie together many of the concepts that up until now may have seemed like a little too much motherhood and apple pie. During this time, we also start to develop the concept of the *inside* versus the *outside* view of an implementation. Packages, especially those using private types, give implementors (the inside view) a way to really develop an abstraction and then to control how this abstraction is used by someone importing this package (the outside view). This distinction is important to make clear for the students.

GENERICS. Another specific feature of the Ada language that may be difficult for some is the use of generics. We find that this may be the most difficult topic to teach and to really grasp. For many, this is their first exposure to the details of writing generics, as opposed to just instantiating and using pre-existing generics. Many come into the class with a couple of notions about generics:

1) Generics are a distant and elegant feature of the language, and
2) They are more work than most project schedules can afford.

There are at least two issues in the generics presentation that must be handled carefully or they can really become points of confusion.

First, is the notion of views. In languages with only global data, there really aren't different views of data for the implementer and the user. When we present this idea in the packaging unit, we can clearly demonstrate that the implementer has full access and full freedom in working with data, whereas the user does not. However, when we get into that area in generics, everything about this perspective seems to turn inside out. The reality is that when it comes to controlling abstractions in generics, the implementer's access and freedom is restricted. The user on the other hand, has more freedom.

The second key point in teaching about generics is that there are a lot of difficult design decisions to be made when developing them. In the case of generic subprograms the designer must be able to distinguish and clearly decide what are to be the generic formal and actual parameters, as opposed to what will be the parameters to the actual generic call. In the case of generic packages, design decisions must be made to determine what should be a generic formal parameter and what should be package data. These, too, can become a point of confusion if not presented slowly and clearly.

Generic formal parameters can really cause students to have to think about design issues in a new light. They now have the option of actually passing subprograms as parameters, another concept that is new to most students. Generic formal constants and generic formal variables, if not handled properly, can cause a student to wonder why they would ever use them, rather than concentrating on what the concepts really involve. There are some very complicated design decisions involved in generic formal parameters.

EXCEPTIONS. Another caution for those doing Ada training, is the appropriate introduction of exceptions. Generally, the use and understanding of exceptions is fairly straightforward. We believe that exceptions are useful and should be employed in Ada designs. However, if they are not introduced early in an Ada course, the chance is high that they will be rarely used. Similar to teaching the IF — ELSE — ENDIF structure in any language, where the else part may or may not be needed, we teach BEGIN — EXCEPTIONS — END right from the beginning. Of course, exceptions may not be used in every executable region, but treating the structure in this way makes them a natural part of these regions. If exceptions are presented as a stand-alone chapter later in the course, with no previous mention, it is more difficult to impress the importance of using them routinely.

Another issue when presenting exceptions is to clearly define the two prevailing schools of thought regarding the use of exceptions. There are those who believe that exceptions should only be used to handle catastrophic unforseen occurrences rather than for predictable events such as trying to pop an empty stack. Both ideas have merit and should be presented equally.

OUTCOME

RESULTS. The success of our Ada program is obvious both in class from the continuously high demand for more offerings, and more importantly, from the subsequent effects on the jobs of participants. Project software reviews and walkthroughs now routinely utilize design review checklists and concepts developed in classes. Class exercises are drawn from realistic examples and actual "real-world" projects. In-house consulting by instructors on Ada contracts is an on-going activity. It is common for participants who completed the courses two or more years ago, to come back with Ada questions or design issues that indicate their daily involvement with concepts taken from the courses. Management continues to issue positive evaluations of participants several months after having completed the courses. In general, the level of Ada awareness and expertise has risen dramatically and the effects of the Ada program can be seen pushing into all layers of the company.

Collins Commercial Avionics, and Collins Avionics and Communications Division, in Cedar Rapids, Iowa, have a proven track record in the area of Ada development. Ada has been used in support of projects for all branches of the military. Additionally, Ada projects comprise a significant portion of our commercial products in the areas of air transport and general aviation. It is estimated that in excess of two million lines of Ada source code have been generated for these and other projects.

CONCLUSION. The experience we have gained in the Ada classroom has allowed us to develop an engineering and management program that has served as a cornerstone for successful transition to disciplined software development. Formal training has enhanced the capabilities of software designers, programmers, and managers, as well as promoted a consistent way of doing business with respect to software, and in dealing with the software crisis.

This Page Intentionally Left Blank

The Ada Apprentice

ABSTRACT - Programming languages are typically taught by the transfer of an almost continuous series of syntactic structures between the instructor and the student, without substance, or meaning, passing through the minds of either. Students are introduced to language constructs in small doses, they then write trivial programs using the newly memorized constructs. Most textbooks reinforce this model by presenting a long sequence of small code segments that ignore any semblance of good programming practices such as structure, documentation, and efficiency.

We, as educators, tend to reinforce this model with an overemphasis on syntax at the expense of creativity. If we want to train computer scientist, and not simply programmers, we must redefine our training mission. Language specific coders are relatively easy to train, however, the development of creative problem solvers is an excruciatingly painful process.

The Ada apprentice model, which has evolved over a period of years, attempts to teach programming skills in much the same way as infants learn to speak a natural language. They are immersed in the process and learn by the observation of, and participation in, correct language constructs.

Learning commences with a goal statement emphasizing creativity and minimizing syntatic regurgitation. Ada is introduced by presenting the students with a correctly functioning program requiring only parameter changes and procedure calls on the second day of class. Programs are then introduced requiring the correction of a series of progressively more subtle bugs. The bug programs are usually designed to demonstrate the consequences of poor programming practices, such as global variables. The flow of furnished programs continues with subprogram specifications and documentation but missing bodies and finally, toward the end of the semester, a required cover to cover program.

The underlying philosophy behind the Ada apprentice model is that language skills, be they Ada or English, are best acquired the same way that small children learn a natural language, by seeing and hearing it done correctly. Classes who have been exposed to this model in a first programming course have had a higher percentage students complete the course and have typically covered three to four additional chapters in the textbook than students using a more tradition learning paradigm. Every indication is that subsequent performance is at least as good as students having been exposed to a more traditional mode.

seven the jury is still out on discipline,
fathers did teach young Urban what it meant to
ill common practice, a large part of the first
nt studying Latin. It was the first time in my
experienced absolute misery. (I hadn't yet
eriously.)

ost embarrassing moments of my entire life
he end of my sophomore year. We were
s Caesar and I was lost as usual. When my turn
las come over to my desk to point out the place
to find that I had written the entire
the Latin. When you are caught in a
this magnitude by your Latin teacher, who also
Dean of Discipline, the term terrifying
ly takes on a whole new dimension.

g my junior year it was suggested that I take
d aspirations of becoming an engineer. The
that much technical writing was done in German.
n was that my advisor, Father Ignatious,
a Mercedes since he was suggesting *the same*
to those desiring to go to medical school, or to
ood.

f one full semester of die der den die I could
complete sentence that native speakers of German
d understand. I likewise could only pick up an
hen these same people were talking among
ordered on humiliation, so I did what any other
een year old youth would do, I gave up German
spirations. I certainly didn't want to become a
field where all they did was sit around reading
No wonder we had to go to war with a people
such an insidious language.

school with a strong belief that I was a good
had after all managed to graduate despite
l record for the number of demerits earned in
t being thrown out. I was even more strongly
didn't posses an ounce of linguistic ability.
concern that I was third in my class, from the
omplished my primary mission in avoiding a full
years.

s later I arrived in Japan for a one year stay
e Sam's army. During my tenure in signal school
e in my class that had selected the far east as
e of duty assignment, there was after all a
g on in Korea. In typical army fashion the

the far east was
e east coast don't
a strong desire to

n buildings. Since
rested in female
re I had to
refutable despite
e enough to speak
vious, I had to
he post USO (kind
e women one
ng a crash course
inadequateness I
ed to be either
re anxiety. At
a public fool of

ours a night, five
e weren't even
the end of three
nd understand what
to anyone that was
great. The
foreign language.

athers has missed?
ame environment
speak. Five year
rmal sentence
hummingbirds who
apable of
d-air.

or several years of
ng language is
rbose programming
hen I realized that
ts I started
ifically how do
without
of time it was
ch patterns. It
reasonable amount
lieve in this
re than one word
of an adult mouth.
ve psychologist is
the language at a

very early age and that structural problems, such as noun pronoun agreement, are simply demonstrations that the rule has not been mastered. There is an extremely important observation to be made if the theory is accepted, namely, if correctly structured language is all that children hear, and subsequently read, they will use the language flawlessly despite the fact that they don't know how to congregate a verb and don't know a split infinitive from a dangling participle.

Now the point of this long diatribe. Can a computer language be learned like small children lean a natural language? I sincerely believe it can and that is what this presentation is all about.

IMPLEMENTATION - To set the tone, for not only the course but for the entire curriculum of study, I start the course with a goal objective. The statement "we are trying to develop computer scientists who happen to program not computer programmers who dabble in computer science" will usually elicit the question "what is the difference between a programmer and a computer scientists." The answer I usually give is "about twenty-five thousand dollars a year." The answer is not quite as cavalier as it may initially sound. I go on to state that a computer scientist is a person capable of creative problem solution that utilize computers while a programmer frequently implements the ideas of others. I believe the distinction is important since most freshman believe that programming is all there is.

The underlying philosophy of my teaching is the belief that if the desire is created the passion will follow. I also believe that the first day of class is critical, give them you best shot to start the class and the rest will flow with ease. I personally think that the typical first day regimen of take attendance, hand out a syllabus, and send them home is a big mistake. It sets the tone for the remainder of the semester. It's hard to instill passion when the instructor trivializes the first class.

On the first day of virtually any computer related course I start by stating that you may thing that you are here to learn how to program (or whatever) but you are not. You are here to learn how to solve problems and coincidentally you are going to use a computer to prove that you have indeed solved the problem.

The following points are emphasized:

1. Problem solving is a generalizable skill.

2. Creativity is first a state of mind and only then a methodology.

3. Program efficiency and creativity draw upon the same skills.

However, simply solving a problem is usually not enough. To set the tone for the remainder of the course, and indeed the remainder of their computer careers, a problem solving paradigm is developed. From this point on when we speak of a solved problem it is implied that the solution is the "best" solution not simply something that works. (You might be surprised if you ask a class of upper-class majors, or graduate students, to describe the characteristics of a 'good' problem solution or 'good' program.)

A 'good' program, like a 'good' problem solution, has three necessary components, namely:

1. Effectiveness. The problem must be solved correctly.

2. Maintainability. The solution must be easily modifiable, especially by someone other than the original problem solver, to reflect changes in conditions and/or deficiencies in the original solution.

3. Efficiency. The solution should be implemented using the minimum amount of resources including human, machine and financial.

The following example is used to illustrate the difference between a solution that works and a good solution. It makes such a point that I have had graduates out in the field for several years call a say they started a new project and the first thing that come to mind was the tennis match problem of many years before. Try giving this problem to your introductory students.

Because of your great mathematical and computer prowess, the activity director of your favorite tennis club asks if you would write a program that could determine the number of matches required given the number of players entered in a single-elimination tournament. What he is requesting is what we computer scientists like to call an algorithm, which is simply a recipe, a step by step procedure that will produce the desired results as a function of the input data.

At this point I typically make sure everyone understands what the term single-elimination means in the context of a tournament. That is, A plays B and C plays D. If A and D each win in round on they play each other in round two and the winner that match is then the winner of the competition since there is no one remaining to be played.

At this point I typically make sure everyone understands what the term single-elimination means in the context of a tournament. That is, A plays B and C plays D. If A and D each win in round on they play each other in round two and the winner that match is then the winner of the competition since there is no one remaining to be played.

A simple, but effective, starting point is to define the problem by giving both the inputs available and the desired output.

Inputs.	1. Fixed number of players (teams if doubles).
	2. Single elimination, when you lose your are out.
	3. One winner.
Output.	1. Total matches required to complete tournament.

To assure that everyone understands the requirements I ask how many matches are required if there are eight players in the tournament. The majority of the group will get the correct answer of seven. What tends to evolve in a problem such as this one is that the group will start to solve the problem by assuming a starting value and working it through, they model the tournament. If they don't I cheat and push them in that direction.

Leading them deeper into the trap I mention that an efficient way to solve a problem such as this is to assume a starting value and work it through recording the steps as we progress. Lets try sixteen as a small manageable number. In the first round there will be eight matches as each of these players square off against each other. Of the original sixteen half survive to go on to the second round. In the second round the eight first round survivors play four matches for the right to continue. These four matches plus the eight from the first round produce a total of twelve matches for the first two rounds. Round three has four players participating in two matches and finally the last round has the two club shills playing each other for bragging rights. The three matches played in the semifinal and final rounds plus the twelve matches required to that point produces a total of fifteen matches. The solution algorithm may be clearer when these results are exhibited in tabular form.

Round	Players Entering	Matches Played	Players Surviving	Total Matches
1	16	8	8	8
2	8	4	4	12
3	4	2	2	14
4	2	1	1	15

The solution appears to be fairly simple. Stated in the form of a recipe it would be:

STEP	OPERATION
1.	Initialize total matches to zero.
2.	Divide the number of players by 2.
3.	Add the result of step two to total matches.
4.	If the number of players remaining is greater than one go to step two.
5.	Report the number of matches.

If you give your friend the activities director a program based upon this algorithm you will come up with egg on your face.

(When reviewing where we went wrong I point out the problem with test data that appears to be different by has identical characteristics.) If the original number of players is not a power of two e.g. 2,4,8,16,etc., an odd number of players will be presented at the start of one or more rounds. Only in mathematics do we have a half player or half match, in the real world one player would be given a bye and automatically proceed to the next round. To accomplish this King Solomon task the algorithm will emerge as follows:

STEP	OPERATION
1.	Initialize total matches to zero.
2.	Divide the number of players by 2.
3.	Add the whole number portion from the result of step two to total matches.
4.	If the result of step two (players remaining) contains a fractional part then round it to the next higher whole number.
5.	If the number of players remaining is greater than one go to step two.

We have now developed an algorithm that will produce the correct answer every time. Does it necessarily follow that this is the best solution to the problem? Not at all. We jumped at the first solution that presented itself. This desire to quickly solve the problem is an occupational hazard that infects most computer programmers that consider themselves programmers rather than problem solvers. The distinction is an important one as we are about to see.

The conceptual approach used to solve the problem was the analysis of how a winner is determined. A simulation of a tournament was then developed. Simply stated the winner must survive until there is no one remaining to play. This required the tracing of rounds from the beginning to the end of the tournament. Is this what we were ask to solve? No we simply took a long route to produce the correct answer which is a simple number.

What do we have to work with? In a mathematical sense we have the set of all players and two subsets: the winner and all losers. If we look at the problem specification again there is a very subtle but key piece of information that is free for the asking although it is not explicitly stated. If there are sixteen players initially and there is but a single winner there must therefore be fifteen losers. In the general case if there are N number of players there are N - 1 losers. Therefore, another track to solve the problem is to analyze how a losers are determined and that is fairly straightforward, they simply do not win a match. Ergo, the number of matches is equal to the number of losers which in turn is equal to the number of players less one. The algorithm now reduces to:

STEP	OPERATION
1.	Total matches is equal to number of players minus one.

Two points require emphasis here. It is obvious that a one step algorithm is more efficient than a five step algorithm, which may repeat itself many times. Mention the subproblems of extracting the fractional and integer parts of a division.

The one-step solution demonstrably optimizes the characteristics of effectiveness, maintainability, and efficiency. The moral of this exercise is do not jump at the first solution that presents itself. Search for solution that is the simplest for the stated objectives. This will usually be the solution that is the most adaptable and, in addition, has the greatest chance of being followed through to its completion.

There is an especially powerful point here for programmers, once coding starts using a particular solution technique almost never will a better approach be taken since it will require scraping existing code.

PROGRAMMING - After the tennis match exercise described above the remainder of the first day is spent describing the characteristics of a 'good' program and the concept of subprograms and parameters. The second day of class witnesses the exposure of the neophyte programmers to a correctly written, well documented, and highly structured Ada program. Appendix A is a listing of program HELLO.ADA. Unnecessary details are hidden in a package STUDENT. Appendix B contains the listing of two procedures from package STUDENT.ADB. The specification of their first program requires that some parameters be changed in procedure calls and the addition of a few calls to existing, although hidden, procedures. Some documentation, conforming to supplied standards, must also be added to the program. A tutorial on the use of the site-specific Ada editor and compiler and the completion of the first programming assignment are both completed during a single one hour lab session. Think of the boost of one's confidence upon producing a functionally correct Ada program that is additionally, highly structured, and well documented on the second day of class.

A series of bug programs are introduced after the students are conformable with modular design and parameter passing. These program have progressively more difficult syntactic and logic bugs that require fixing. I originally learned Pascal by debugging student program and can personally attest to the effectiveness of this technique. The bug programs additionally have errors in program structure, rather than logic, such as code within a loop that produces the same answer every iteration. Segments of required documentation also require completion.

An early bug program demonstrates the undesirable side effect that can be caused by unintentionally changing a global variable. The negative effects of poor programming practices are dramatically demonstrated when the consequences appear right before your eyes. To paraphrase Turgenev who paraphrased an old Chinese proverb, "A bug program shows me at a glance what it takes dozens of pages of a book to expound."

A problem inherent in the teaching of a first programming courses is the large diversity of ability and actual programming knowledge. Some students are being exposed to programming for the first time while others have had a substantial amount of prior programming. I tell the assembled mass on the first day of class that the students that have previous programming experience have only a slight disadvantage over those with no experience. It takes time to undue all those poor habits.

This model allows for differences in student's ability level and motivation by always assigning extra credit portions to programming specifications. Students are additionally encouraged to look into the supplied package body to see how the "tricks" are done. The highly motivated students will be using techniques from the package by the third program.

A series of assignments then build upon existing programs. One of the bug programs is designed to demonstrate accumulation by requiring the input of a series of integers. Modification of the program requires that an array be used to store the supplied numbers. The subsequent modification of the array program requires the inclusion of a sort procedure. No only does this technique reduce typing time it is valuable experience in program maintenance, which is where most of them will start the careers.

The progression continues by giving the students a program with a procedure heading, declaratives and documentation. They must supply the statements necessary to complete the body. There are subtle lessons in this approach. The use of supplied data objects having meaningful names has far greater impact than a dozen lectures describing the "real world" disadvantage of cryptic data names. We educators have tended to ignored our own experience that repeatedly demonstrates that we learn best by doing, not by listening to someone else tell us how the task is accomplished. The initial series of programs are designed to introduce the software engineering concepts of life cycle, data abstraction and procedural abstraction. The general conceptual premise is the student's immersion in good programming practices. They do not have the flexibility to develop poor habits.

It has been my experience that students have a preconceived idea of approximately how much external class time they will devote to any course. In a programming course keyboard time counts. By giving students the vast majority of required code the time spent on the terminal is quality productive time.

CONCLUSION - The Ada apprentice model has evolved over a period spanning ten years of teaching programming intensive courses ranging in levels from introductory to graduate. It is my sincere belief that we don't teach students anything we only help them learn. If we as educators create a conducive and stimulating environment the students will supply the required cerebral dexterity. The maximum programming grade given for a program submitted on the second day of class is more inspirational than a thousand admonishments that "this is a tough course requiring substantial time."

It is difficult objectively evaluating this method without having the same instructor teach parallel courses using two different techniques. There would even be flaws in such a technique. When I have taught using this method when other instructors have used a more traditional approach my students have completed more programs, covered more material, and have received higher grades in the next course in the sequence taught by someone other than myself. Pedagogically I believe that immersion encourages internalization more effectively than memorization. The shift of class time to substance from syntax transfers the emphasis from simply creating programs that work to the creation of "good programs" that creatively solve problems with the best solution.

The apprentice model, true to its name, allows students to take progressively more responsibility for their craft. The progression is initiated by observing the works of a master craftsman. Small finishing touches are initially added by the novice. The apprentice supplies progressively more detail, under the watchful eyes of the master. Finally that magical day arrives when the students reaches journeyman status and enters the wonderful world of data structures.

Appendix A

```
--      This program is designed as a simple demonstration in
--  introductory programming concepts. Documentation in all
--  sections must be completed before submitting finished
--  version. Documentation must conform to the macro programming
--  standards.

with STUDENT; use STUDENT; -- Tools for student programs.

procedure HELLO is

  procedure GO_TO_WORK is
  --  Purpose: To performs the work for program HELLO.
  --  Requires: Nothing.
  --  Modifies: The terminal screen.
  --  Description: This subprogram is a general purpose procedure
  --  designed to demonstrate parameter passing.

  begin - procedure GO_TO_WORK
    DISPLAY_MESSAGE("Hello World");
    -- Pass your name as the actual parameter in the next line
    DISPLAY_MESSAGE(" my name is Urb LeJeune");
    New_line;
    DISPLAY_MESSAGE(
      "Send help I'm a prisoner inside this program");
    New_line(2); -- Send two CR LF characters
    DISPLAY_CHARACTER('&',10); --Display the ampersand ten times
    DISPLAY_CHARACTER('?');     --Display one question mark.
    -- Note that the default replication factor is one.
    DISPLAY_CHARACTER('*',5);  -- Display the asterisk five times
    New_line(3); -- Send three CR LF characters
    DISPLAY_MESSAGE("end of job");
  end GO_TO_WORK;

begin -- procedure HELLO
  GO_TO_WORK; -- Call procedure GO_TO_WORK
 -- All student programs should end with a call to End_Of_Job
  END_OF_JOB;
end HELLO.
```

Appendix B

```
package body STUDENT is
  -- Purpose: A group of tools for student programmers

  procedure DISPLAY_CHARACTER(
    WHAT  : in character;
    COUNT : in positive := 1 ) is
  --Purpose: To display the passed character a repetitive number
  --  of times.
  --Requires: WHAT passed as the character to be displayed
  --          COUNT passed as the replication factor. It must be
  --          a positive number which defaults to 1.
  --Modifies: Display character(s) at current cursor position and
  -- the cursor is left at next available display position.
  --Description: Procedure DISPLAY_CHARACTER displays a passed
  -- character WHAT COUNT number of times.

    TEMP_STRING : TEXT(80);

  begin -- procedure DISPLAY_CHARACTER
    SET(TEMP_STRING,""); -- Build display string.
    for INDEX in 1..COUNT loop
      APPEND(WHAT,TEMP_STRING);
    end loop;
    Text_io.Put(VALUE(TEMP_STRING));
    if PRINTER_IS_ON then Put(PRINTER,VALUE(TEMP_STRING));
    end if;
  end DISPLAY_CHARACTER;

  procedure DISPLAY_MESSAGE(
    MESSAGE    : in string;
    BACKGROUND : in boolean := true) is
  --Purpose: To display the contents of the passed string.
  --Requires: MESSAGE is any valid string.
  --          BACKGROUND is a boolean switch that will cause
  --          MESSAGE to display in background if TRUE and
  --          foreground if false. The default is true.
  --Modifies: Message is displayed at current cursor position and
  -- the cursor is left at next available display position.

    ROW, COL : integer;
    TEXT_1   : text(80);

  begin -- DISPLAY_MESSAGE
    GET_POSITION(ROW,COL);
    Put(ROW,COL,MESSAGE,REVERSE_VIDEO => background);
    if PRINTER_IS_ON then Put(PRINTER,MESSAGE);
    end if;
  end DISPLAY_MESSAGE;

end STUDENT;
```

A Sequence of Freshman Level Integrated Laboratory Assignments

John Beidler
Computing Sciences
University Of Scranton
Scranton, PA 18510

BEIDLER@JAGUAR.UOFS.EDU
(717) 941-7446
(717) 941-4250(FAX)

1. Introduction

There is no doubt that the programming language plays a significant role in the early courses in computing. In fact, A primary obligation of course instructors must be to keep the students from getting so wrapped up in the programming language that they miss the transcending software development concepts. When selecting a programming language to support the first several courses in computing we use the following criteria:

a. In the worst case, the language should not get in the way of teaching concepts.

b. In the best case, the language should provide direct support to important concepts.

In following these criteria, in 1976 our department selected PASCAL as our primary support language, in 1984 we moved to Modula-2, and went to Ada in 1990. With only one year's experience with Ada, there is little doubt about the positive and immediate impact this choice has had on our curriculum. This paper explores one aspect of a freshmen level course that uses the packaging capability of Ada to supply support for laboratory assignments.

However, we must be cautious, we must not let students become so enamored with the programming language that they miss the overriding concepts. The teacher must supply the balance. This balance is partially achieved through a variety of software development experiences where each experience has a significant software development goal. The software development experiences roughly fall into three categories:

1. Laboratory Assignments.

2. Traditional Programming Assignments.

3. Projects.

Projects may be individual or group projects that are performed over an extended time frames of at least a third of a semester. Projects must have a substantial non-coding components and include significant analysis and design experience and documentation. Traditional assignments emphasize coding, testing, and debugging, and may include some analysis and design.

Laboratory assignments, the object of this paper, are designed to be complete in a 1.5 to 2 hour laboratory period. There primary purpose is to demonstrate hardware facilities or specific relationships between programming concepts and programming language features. Laboratory assignments should be designed to provide insight. Because of the one and one half to two hour time frame of laboratory periods laboratory assignments should not be designed to provide analysis and design experiences.

This paper describes a coordinated series of laboratory assignments that accompany our CMPS 144 (CS 2) Course. These assignments were designed to demonstrate relationships that exist between programming concepts and programming languages features. This course is a broad-based introduction to computer science that uses Ada as its support programming language. This course begins with an emphasis on analysis and design. This means that packages and subprograms are formally presented early in the course, before control structures and data structures. This strategy has the advantage of encouraging good software design by emphasizing encapsulization, top-down design and stepwise refinement. However, this approach requires resources to support it.

Currently, there is a trend in computer science education towards the coordination of science-like laboratories in support of various computing courses. As this trend continues, several issues regarding laboratory assignments must be addressed. Two important issues are:

1. Laboratory assignments must be clearly defined with appropriate objectives and have reasonable expectations that the majority of students should be expected to complete the assignment during a normal laboratory period (1.5 to two hours).

2. Each laboratory assignment must have a clearly defined starting points, statement of work, and appropriate resources that allow the students to concentrate their efforts on laboratory assignment's goal and not waste time on peripheral issues.

Many science laboratory assignments have two objects, the stated objective and the transcendent objective. The stated objective is the operational goal of the laboratory assignment. The transcendent objective is the educational objective that the students are expected to observe through the laboratory assignment.

Setting up a computing laboratory assignment should be more that giving a description of the assignment and a workstation. A laboratory assignment should not be a race to grind out the necessary code within a given period of time. A laboratory assignment should clearly state the starting point and directions on how to proceed in accomplishing the assignment, possibly even partially complete source code, so that the students have a good opportunity to observe the transcendent objective while working on the stated objectives.

One method of accomplishing this is with a coordinated series of laboratory assignments, all having the same, or similar, stated objectives but each with a distinct, or unique, transcendent objective. The series of assignments described in this paper all have the same stated objective, namely, build a software system that draws the symbols in your user I.D. on the terminal using stick figures, as illustrated in Figure 1. By varying the approach to accomplishing the stated objective each assignment targets a unique transcendent objective. By seeing a variety of software concepts and their implementations in the context a particular problem they have a framework for evaluating the relative capabilities and merits of various approaches.

```
X       X  Y       Y  ZZZZZZZZZ        4
 X     X    Y     Y          Z   4     4
  X   X      Y   Y          Z    4     4
   X X        Y Y          Z     4     4
    X          Y          Z      444444444
   X X         Y         Z             4
  X   X        Y        Z              4
 X     X       Y       Z               4
X       X      Y      ZZZZZZZZZ        4
```

Figure 1 Example Login

Section 2 of this paper describes our primary resource, a package we refer to as DOODLE. Section 3 describes the series of DOODLE-related laboratory assignments and their relationship to the material covered in the course. Section 4 summaries our observations about the use of coordinated laboratory assignments and their relationships to material covered in the courses.

2. The DOODLE Package

The DOODLE package evolved from a Modula-2 library module that supported the same course in previous years. However, when the change was made from Modula-2 to Ada, the module was not just translated into Ada. Rather, the module was redesigned to take advantage of the features available in Ada. The specifications for the DOODLE package appear in Figure 2.

The DOODLE package makes a limited collection of resources visible to users. These resources support character graphics on a standard terminal. Currently versions of this package exist for the VAX, SUN, and MS-DOS environments. Its basic resources are:

a. CLEAR_SCREEN -- A procedure that erases the terminal screen and places the cursor in the upper left hand corner.

b. DRAW_LINE -- A procedure that draws a "best fit" line from the coordinate position indicated by the first two coordinates to the position indicated by the second two coordinates using the character indicated by the fifth parameter. This procedure uses a well known graphic algorithm for drawing straight lines.

c. GET_INT -- A data entry procedure that displays a prompting message on the screen, accept an integer as input, and returns an integer. The original Modula-2 version of this procedure used a pop-up window for the dialogue. An X-Windows version is under development.

d. GET_CHAR -- A single character data entry equivalent of GET_INT.

```
package DOODLE is

  subtype ROW_TYPE    is integer range 1 .. 23;
  subtype COLUMN_TYPE is integer range 1 .. 80;

  procedure CLEAR_SCREEN ;
      --------------------------------------------------------------
      -- Pre-Cond: None
      -- Post-Cond: Erase screen, position cursor at (1,1)
      -- Exceptions Raised: None
      --------------------------------------------------------------

  procedure DRAW_LINE
      ( LEFT_ROW     : in     ROW_TYPE ;
        LEFT_COLUMN  : in     COLUMN_TYPE ;
        RIGHT_ROW    : in     ROW_TYPE ;
        RIGHT_COLUMN : in     COLUMN_TYPE ;
        THE_SYMBOL   : in     character  ) ;
      --------------------------------------------------------------
      -- Pre-Cond: THE_SYMBOL must be a display character
      -- Post-Cond: Draw a "best-fit" straight line between the
      --           indicated screen coordinates using THE_SYMBOL
      -- Exceptions Raised: constraint_error
      --------------------------------------------------------------

  procedure GET_INT
      ( PROMPT    : in     string ;
        THE_ENTRY :    out integer ) ;
      --------------------------------------------------------------
      -- Pre-Cond: PROMPT is a string of display characters
      -- Post-Cond: Place integer reply into THE_ENTRY
      -- Exceptions Raised: constraint_error
      --------------------------------------------------------------

  procedure GET_CHAR
      ( PROMPT    : in     string ;
        THE_ENTRY :    out character ) ;
      --------------------------------------------------------------
      -- Pre-Cond: PROMPT is a string of display characters
      -- Post-Cond: Place character reply into THE_ENTRY
      -- Exceptions Raised: constraint_error
      --------------------------------------------------------------

end DOODLE;
```

Listing 1 DOODLE Specifications

A few additional items are encapsulated in the package, but these four procedures are the basic support upon which the coordinated set of laboratory assignments are constructed.

3. The Coordinated Laboratory Assignments

1. Build a program using the DOODLE package as a resource to draw with character graphics on a VT-100 compatible terminal your user ID with stick figures. Each character should appear in a box that is 14 columns wide and 9 rows deep.

2. Reorganize #1 emphasizing procedural abstraction by creating a procedure for each symbol in your user ID and placing the code that draws each symbol into the appropriate procedure. Give each procedure a menaingful name, stating what the procedure does, like DRAW_AN_X.

3. Place three formal parameters in the declaration of each procedure in #2. The three parameters are for passing information between the procedure and calls to the procedure. The three pieces of information are the left row and top column of the rectangle where the symbol is to be placed, and **character** used to draw the figure. To verify that the parameters are being used correctly, make several calls to each procedure modify the parameters to draw the figure in various locations of the screen. For example, if the formal parameters are,

```
procedure DRAW_AN_X (
     X_ROW : in    ROW_TYPE;
     X_COL : in    ROW_TYPE;
     X_SYM : in    character);
```

then a call to DRAW_LINE might appear as,

```
DOODLE.DRAW_LINE (X_ROW+0, X_COL+0,
                  X_ROW+9, X_COL+14, X_SYM)
```

where the first four parameters are the sums of one of the screen coordinates of the upper left of the box containing the figure and a relative coordinate of the line in a 14 by 9 box.

This laboratory assignment is critical because of its use in later assignments. Verify the assignment by modifying, recompiling, and executing the program several times, each time modifying the actual parameters,

```
DRAW_X (1, 1, '*');
DRAW_X (5, 40, '?');
```

and drawing the symbols in different areas of the screen.

Table 1 Assignments 1-3

The original purpose for the DOODLE package was to support a collection of laboratory assignments whose purpose was to promote an early emphasis of procedures and parameters in the course. We realized that we could use this collection of assignments as the foundation for a more complete series of laboratory exercises. This collection of exercises centered around the single simple visual problem, illustrated in Figure 1, and uses that problem to demonstrate various relationships between programming concepts and programming language support. They also demonstrate the value of an Ada Programming Support Environment.

The stated objective behind the assignments is the development of a program that draws the user's system ID on the text screen with stick figure graphics. These assignment have several advantages, including the immediate feedback, on the screen, of the display of the lines that form the stick figures for the symbols in the user's system ID. Using the visual feedback, most students can build a working version of the first assignment in about an hour and a half.

4. Remove the formal parameters from the procedures build in #3 and replace them with three variables defined within the procedure. Now, within the procedure make two calls to DOODLE.GET_INT to get the row and column coordinates of the upper left during program execution and call DOODLE.GET_CHAR to get the symbol to be used.

5. Note in #4 with the dialogues you had to do the same thing over and over again to get the data for each symbol. Write, and correctly place one procedure,

```
procedure DIALOGUE (ROW :      out ROW_TYPE;
                    COL :      out COL_TYPE;
                    SYM :      out character;
                    MSG : in       string);
```

that will be used by all of the procedures to perform their dialogues. The code for this procedure should basically be one copy of the data entry dialogue code from #4. The MSG is concatenated to the stings passed to GET_INT and GET_CHAR, as in

```
DOODLE.GET_INT ("Enter the row for " & MSG, ROW);
```

Call this procedure for all sets of data entry.

6. Place exception handling into the DIALOGUE procedure in #5 so that each data entry is handled by its own exception handler. This is accomplished by placing each data entry into its own statement block,

```
begin
   ...
  exception
   ...
end
```

A **constraint_error** is raised by the system if the data entry is correct, or you may raise it within an **if** structure if the row or column entered would force the symbol to go off the screen.

Table 2 Assignments 4 - 6

Typically, students complete each laboratory assignment in about one to two hours, the time of a typical laboratory period. Tables 1 through 5 briefly outline the assignments. The first assignment gives students an opportunity to become acquainted with the programming support

7. Starting with #3 replace the three parameters by one parameter, a record containing the three pieces of data,

```
procedure DRAW_X (X: in    SYM_RECORD);
```

Initialize each record using an aggregate and use the data in the record using "dot" notation,

```
DOODLE.DRAW_LINE (X.ROW+0, X.COL+0, X.ROW+9, X.COL+14, X.SYM)
```

8. Starting with #7, take a look at the procedures you have to draw each symbol. There is a sameness to them. We are now going to get rid of all of them and replace them by one procedure,

```
procedure DRAW_SYMBOL (THE_SYMBOL : in    SYM_REC ;
                       THE_LINES  : in    SYM_ARRAY ;
                       NO_OF_LINES: in    integer);
```

where NO_OF_LINES is the number of lines required to draw the symbol. The line information is stored in a structure defined as:

```
type SYM_REC is record    LEFT_ROW : ROW_TYPE;
                          LEFT_COL : COL_TYPE;
                          RIGHT_ROW: ROW_TYPE;
                          RIGHT_COL: COL_TYPE
                end record;
type SYM_ARRAY is array (1..15) of SYM_REC;
```

With these declarations the information to draw an X is placed in the first two locations in the array of records,

```
X_LINE : SYM_ARRAY ;

X_LINE(1) := (0, 0, 8, 14);
X_LINE(2) := (0, 14, 8, 0);
```

and the X would be drawn with a call to DRAW_SYMBOL,

```
DRAW_SYMBOL (X, X_LINE, 2);
```

Within DRAW_SYMBOL, the symbol is drawn with a **for** loop,

```
for INDEX in 1..NO_OF_LINES loop
     DOODLE.DRAW_LINE (
          X.ROW + XLINE(INDEX).LEFT_ROW,
          X.COL + XLINE(INDEX).LEFT_COL,
          X.ROW + XLINE(INDEX).RIGHT_ROW,
          X.COL + XLINE(INDEX).RIGHT_COL, X.SYM);
end loop;
```

Table 3 Assignments 7 - 8

9. In #8, replace the definition of SYM_ARRAY with an unconstrained definition. Then define each symbol's lines with its appropriate constraints,

```
X_LINE : SYM_ARRAY(2) ;
```

Redefine DRAW_SYMBOL with only two parameters and use an attribute function to obtain the upper bound on the **for** loop.

10. Start with #6 and modify the exception handler by placing each data entry in a **while** loop where the loop terminates only when a valid data item is entered,

```
VALID_DATA := false;
while NOT VALID_DATA loop
     begin

        exception

     end
end loop;
```

11. From an object oriented point of view, all of the information that belongs together should be kept together. Start with #9 and include the array of line information in the SYM_REC. Define SYM_REC as a record with a discriminant and use the discriminant to constrain the array in the record,

```
type SYM_REC (SIZE : positive) is
     record
        ROW  : ROW_TYPE ;
        COL  : COL_TYPE ;
        SYM  : character;
        LINE : SYM_ARRAY (SIZE) ;
     end record;
```

12. Modify the exception handlers in #10 so that a user has a fixed number of attempts, like 5, to enter correct data before the program is simply aborted. This requires additional code after the **while** loop containing the exception handler, an **if** structure to determine the reason the loop terminated and handle it accordingly.

Table 4 Assignments 9 - 12

environment -- the editor, compiling, linking, and running programs. The goal of the second assignment is to acquaint students with procedural abstraction. The third assignment introduces data flow through procedures. The fourth assignment acquaints students with interactive IO. The fifth assignment encapsulates the interactive IO of the fourth assignment with a single procedure.

Assignment Six uses the interactive IO developed in Assignments Four and Five as the context for demonstrating exception handling. The students build three exception handling blocks around the three data entries in the DIALOGUE procedure.

Assignment Seven emphasized the representation of objects and their attributes. Each rectangle may be viewed as an object. In this assignment, the objects are partially encapsulated by using a record to bring together three values that combine to define the location and appearance of the object on the terminal screen.

Assignment Eight is a pivotal assignment. It demonstrates the classical trade-off of data structures between algorithms and the representation of information. Specifically, with this assignment the students see that by storing the information to draw each symbol in an array. The immediate impact of this use of a data structure is that all the separate drawing procedures for each symbol are replaced by one procedure.

Assignment Nine introduces unconstrained arrays. Assignment Ten returns to the exception handling in Assignment Six and has the student experiment with more sophisticated exception handling. Assignment Eleven extends Assignment Nine by continuing the encapsulation of objects by including a constrained array in the records that represent objects. This introduces the students to records with discriminants.

13.	Replace the exception handlers in #6 by a recursive exception handling procedures by placing each data entry in its own recursive procedure, contained in `DIALOGUE`. Each procedure handles possible exceptions by recursively calling itself.
14.	Merge #11 with #12 or #13. The fundamental problem you encounter is due to the change in #12 and #13. The simplest way to merge these is to pay attention to procedure names. Specifically, the `DRAW_SYM` procedure, as its name implies, draws the symbol. The name does not indicate that a call to the `DIALOGUE` procedure should be made from within the procedure. Instead, make the apporpriate call to the `DIALOGUE` procedure, to obtain the necessary information, before each call to `DRAW_SYM`.

Table 5 Assignments 13 - 14

Assignments Twelve and Thirteen continue the experimenting with exception handling starting with Assignment Six and continued in Assignment Ten.

Assignment Fourteen wraps everything up by merging together the object encapsulation and exception handling threads of several previous assignments into a single complete piece of software.

Although there are fourteen assignments in this sequence, not all of these assignments are used in any one semester. These assignments may be interleaved with other laboratory assignments. The selection of laboratory assignments may depend upon a balance in the material emphasized in laboratory assignments, programming assignments and projects. Figure 2 illustrates the possible sequencing of these assignments.

4. Experience with DOODLE-based Assignments

This series of assignments stem from a desire to introduce subprograms early in the semester. Specifically, we wished to have a resource that provides an interesting set of assignments early in the semester, which emphasize top-down design before control structures and data structures have been introduced. The use of graphics and its advantage of immediate feedback was attractive. However, the changing of screen modes between graphics and text is problematic on some systems. This led naturally to the use of character graphics with the screen in text mode.

A coordinated series of laboratory assignments was first used during the 1989-90 academic year at the University of Limerick (Ireland). The support language in the first year courses at The University of Limerick is Modula-2. A Modula-2 support module was constructed for the JPI's Modula-2 programming environment under MS-DOS. The JPI environment had several advantages, including windowing support. At the University of Limerick the course, equivalent to CS 1 and CS 2 combined, was taught over three ten week terms. Because of limitations within Modula-2, those assignments involving unconstrained arrays and exceptions were not included. However, there was an additional assignment in which a generic lists package was used to store the line segment information for drawing the figures.

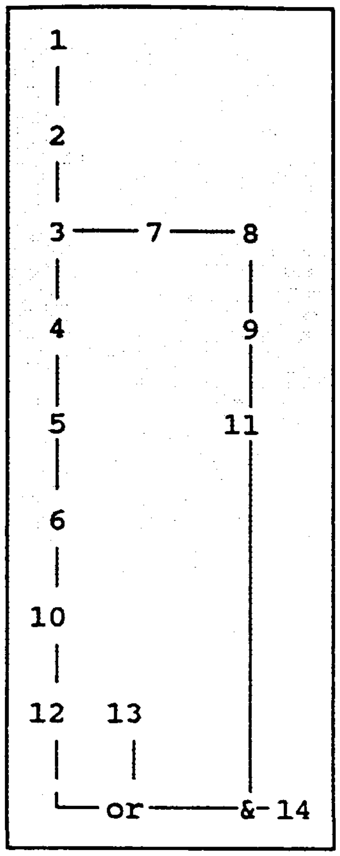


Figure 2 Assignment Sequencing

During the 1990-91 academic year, an equivalent Ada support package was developed, first using the MS-DOS based Meridian Ada system, then a VT-100 version was developed to work on the VAX under VMS and on the SUNs, running SUN OS under X-Windows. Seven of these assignments were used during the Spring Semester in our CS 2 course. These assignments were interleaved with other assignments that acquainted the students with other resources and the bounds of these resources.

5. Conclusions

There are two major advantages in having a coordinated series of laboratory assignments. First, students gain a greater appreciation of conceptual issues with less time wasted in starting each assignment. Specifically, since less time is spent starting up subsequent related laboratory assignments students have a more productive laboratory experience. Second, the single stated objective gives the students a single framework for measuring the value of various programming concepts and the corresponding language support.

When interviewed at the end of the semester about there experiences with Ada versus their previous programming language experience, to a person the students felt much more positive about Ada, and they all provided reasons for their choices. In many cases there reasons relate to the sequence of assignments described in this paper.

Currently we use Pascal to support the first course and Ada to support the second. We are heading towards science-like laboratory support for both of these courses. We believe that well designed laboratory experiments will dramatically improve the educational value of these courses. However, the full value of laboratory support requires more than good text books and laboratory manuals. They require Instructor manuals to guide faculty unfamiliar with this approach, its possible goals and limitations.

This Page Intentionally Left Blank

Educational Significance of a Declarative Ada Dialect

Paul A. Bailes
Dan Johnston
Eric Salzman
Li Wang

Language Design Laboratory
Key Centre for Software Technology
Department of Computer Science
University of Queensland QLD 4072
AUSTRALIA

ABSTRACT

The essential affinity between software development and established engineering activities leads to the supposition that software engineering education should follow engineering traditions. Notable among these is a strong foundation in a general vocbaulary of the relevant disciplines as well as the teaching of the best professional practice through what amounts to good example. For software engineering, the increased expressiveness of functional languages allows a more extensive vocabulary of programming concepts to be explained. Also, the simple mathematical structure of functional programs allows the idea that programs can be developed to mathematical quality standards to be converyed with credibility. Unfortunately, the apparent radical novelty posed by functional programming and languages inhibits their adoption and consequent enjoyment of these benefits. Our solution is to introduce functional programming into existing cultures through preprocessor-extensions of familiar languages to support the functional paradigm. Ada is our choice for a culture where our efforts may yield most significant benefits. Our DAD preprocessor represents a convincing blend of Ada style with the terseness and mathematical tractability of "genuine" functional languages.

INTRODUCTION

We expose two complementary needs in software engineering - education based on the credible application of formal methods, and improved access to new software concepts such as these from within established technology cultures - and demonstrate how a preprocessor-based development of Ada satisfies them. Our DAD (Declarative Ada Dialect) significantly extends previous work in extending Ada for functional programming [1], and provides expressiveness actually quite beyond that usually expected of "pure" functional languages.

FUNCTIONAL PROGRAMMING AS A BASIS FOR SOFTWARE ENGINEERING EDUCATION

Our premise is that Software Engineering (SE) is essentially similar to established branches of Engineering (Electrical, Mechanical, Civil, etc.), and that consequently, SE Education (SEE) should be modelled on the essential paradigms of traditional engineering education. Furthermore, when these paradigms are instantiated with the specifics of software development, the special role of Functional Programming (FP) becomes evident.

Software Engineering is Engineering

Some skeptics are confused by the absence of imposing physical results from the software development process. The simple answer is that the "Engineering" concept is about processes, not the form of result. Were the contrary true, how could the wide diversity in the results of the engineering process (soaring towers to miniature circuits) be accepted. Indeed, given that we accept this diversity as "Engineering", the extension to non-physical resulting software artefacts seems absolutely unexceptionable.

More directly, just because somebody produces an artefact that an engineer might, does not make what that somebody has been doing into "Engineering". We assert that any process of construction that is to be applied with "Engineering" disciplines of quality- and cost-control is a legitimate form of engineering.

The (Software) Engineering Education Paradigm

Inspection of typical Engineering curricula (at least within our own environment [2]) leads to the identification of three general characteristics:

(1) the devotion of the early parts of the curriculum to basic science and math;

(2) the inclusion of non-technical, "professional" content (management, economics, etc.);

(3) the transition from abstract basics to large-scale applied project work by the end of the curriculum.

Introductory (Software) Engineeering

What basic science and math are appropriate for the introductory education of (software) engineers? We respond with the question of what is the purpose of this foundational material? We propose the following answers:

(a) Foundations are not intended to provide the content that will be applied in the initial professional placement after graduation. The project courses that dominate the later stages of engineering curricula are where knowledge of "real world" tools and techniques are acquired.

(b) Foundations do provide a useful general conceptual vocabulary that can be used for many purposes, not least of which is a critical appraisal of the capabilities of the "real world" tools and techniques to be met subsequently.

(c) This initial provision of a powerful vocabulary represents the germ of the most significant role of the foundation: to provide ("indoctrinate", if you will) a correct "engineering" attitude (in the abovementioned terms quality- and cost-sensitivity) toward the process of (software) construction, as opposed to the mere recitation of the specifics of contemporary technological fads;

(d) Despite the separation in principle of foundations from applications, sensible pedagogy will accommodate a stimulating applications flavour.

Functional Programming has the Answers

Corresponding to each of the above criteria for foundational material, FP responds as follows.

(a) An unnecessary but important benefit is that FP is an important SE skill. The literature on software failures (typified by [3]) is replete with accounts of software that works but according to erroneous specifications. Rapid Prototyping [4] offers the best prospect for specification validation by allowing SEs to present their understanding of a customer's requirements in the customer-oriented terms of a working model. Functional Languages that emphasise expressiveness and simplicity of programming at the possible expense of execution performance are ideally suited to Rapid Prototyping [5, 6]

(b) Functional languages are simply more expressive than conventional procedural/imperative languages. Granted, all "programming" languages are in some sense equivalent (by the Church-Turing thesis), but this equivalence is not in a sense related to the usefulness of languages in user- (i.e., programmer-) oriented terms. The effective expressiveness of a language is determined by the richness of the set of linguistic constructs that it supports. The relative richness of functional languages is demonstrable from theoretical and pragmatic perspectives, as follows.

Theoretically, functional languages are more expressively complete [7]. Not only do they tend to have more powerful inbuilt data structuring mechanisms (e.g. lists vs. arrays, polymorphism), but their user-definable higher-order functions allow their extension by simple declaration to accommodate new paradigms of control and data structuring as they appear. In the educational setting, this means that a need to describe in concrete some/any programming concept is more readily satisfied with a functional language.

Pragmatically, the case is closed by the example of how languages are explained. Denotational [8] definitions/explanations of languages proceed in essence by translating their programs into the equivalent functional programs. Functional programs are sometimes translated into procedural programs for implementation purposes, but the point is that when explaining programming concepts (as in an educational setting), functional programming is a style of choice.

(c) We identify the quality/correctness issue as the important attitude issue in the intellectual formation of software engineers. The idea that software can in fact be created according to the mathematically-correct standards that prevail in other engineering fields is especially important to convey early, given the quality-hostile environment created by many programming tools/languages. Functional programs are demonstrably more amenable to formal reasoning than conventional languages.

Granted, formal methods may not necessarily become industrially-prevalent for a long time, if ever. However, it is important to provide students with an ideal standard by which to measure the support for correctness-achievement offered by "realistic" tools.

Note that cost/efficiency issues, while not featuring so prominently in this derivation, are by no means cast aside. True, changes in hardware costs and performances over the years means that absolute statements about what is "efficient" and what isn't are not possible, but broad complexity issues (e.g. linear vs. polynomial vs. exponential algorithms) are perfectly feasibly presented from within functional languages.

(d) Because functional languages are both richer and simpler than conventional languages, the feasibility of motivating, ambitious applications programming/prototyping is in fact increased.

Resources for Functional Programming

To teach FP successfully, we need (in addition to personnel) supporting materials in the way of language implementations and texts. Many high-quality implementations of functional languages exist that support the above policy to varying degrees:

Hope	ftp from brolga.cc.uq.oz.au - /pub/hope
Miranda*	e-mail to mira-request@ukc.ac.uk
ML	ftp from research.att.com - dist/ml
Scheme	read Internet news group comp.lang.scheme

but for reasons to be explained in detail below, they fail to completely satisfy curriculum needs.

Many books about FP are available, but only some are suitable for introductory teaching. Two are especially worthy of mention.

- *Structure and Interpretation of Computer Programs* [9] emphasises the development of a powerful initial vocabulary of programming concepts, and how many common procedural concepts are represented in functional terms.
- *Introduction to Functional Programming* [10] emphasises formal methods of program development, without detracting too much from the range of examples and applications presented.

WHY AN ADA DIALECT?

Accessing Functional Technology

The strength of functional languages - their support for an idealised view of programming - is their practical downfall. They differ from conventional languages at user and machine levels. User-level differences mean that potential users are intimidated by unfamiliar syntax. Machine-level differences preclude the integration into complete systems of a mixture of components written in functional and conventional languages (as in an incremental prototyping situation). These drawbacks are perhaps less intrinsically-influential in an educational (compared to an industrial) setting, but the residual and transmitted problems remain significant:

- project courses which attempt to recreate industrial scenarios will suffer from the industrial drawback of non-integrability of functional and conventional components;
- in spite of our earlier arguments, anything that detracts from the industrial credibility of a tool doesn't help its acceptability for introductory teaching, at least with some (influential) faculty;

* "Miranda" is a trademark of Research Software Ltd.

- ditto for tools that can't be used in later stages of the curriculum;
- some faculty will resist learning too many new languages.

In summary, even though good implementations of modern functional languages are now readily accessible, the effective accessibility of functional programming techniques requires their integration into mainstream language cultures.

Influencing Practice through Education

Educational application is the best (only?) way to seriously market tools and concepts to their prospective users. Dichotomies between ideal tools for new concepts and the "real" tools used in practice can actually act as an advertisement against the new concepts.

Why Ada?

All the above is predicated upon the existence of a fixed, non-functional language culture into which functional techniques have to be inserted. Ada represents such as culture *par excellence*. The industrial/professional situation needs no further exploration, but in the educational sphere that situation is now being replicated. The pressure for wider adoption of Ada as the "standard" teaching language (obviously in support of the Ada professional culture) [11] means that the credibility and acceptability of functional programming will be even more significantly enhanced by its integration with Ada, and inherited support from Ada support tools.

This is the scenario to which DAD in intended to contribute, as suggested by its very name as a complement to Ada, in both practice and teaching.

Preprocessor Implementation

Three factors influence the realisation of an Ada-flavoured functional language:

(1) the need for the integration of DAD components and Ada components;

(2) the need to stage the development of DAD itself, spiralling up to successively closer approximations to the functional ideal;

(3) the consequent opportunity to manifest a variety of DADs, from which an appropriate selection can be made to suit the various stages of the curriculum. We would begin with the highest level that is closest to the functional ideal, and as project work becomes more "real-world"-oriented, work down the hierarchy more towards "real" Ada.

Factor (1) influences us heavily toward a preprocessor implementation. Factors (2) and (3) are compatible with an series of preprocessors, each transforming programs at one level of the DAD hierarchy to the next below. Finally, the conceptual accessibility represented by the design of the language and the staged implementation is complemented by pragmatic accessibility in the portability of supporting software, which is readily achieved by writing the preprocessor(s) in Ada itself.

DAD IN DETAIL

We briefly traverse the hierarchy. See the subsequent EXAMPLES AND APPLICATION section for more illustration.

First-class Functions

"Raw" Ada actually possesses the defining characteristic of functional languages - first-class procedural abstractions - albeit disguised as tasks. Genuine higher-order functions are therefore achieved by building an applicative interface to tasking, as follows.

Function Type

Declare:

> **type** function_type **is function** [formal_part] **return** result_type;

so that *function_type* is the class of functions with arguments *formal_part* and result *result_type*, which can be another function_type.

Function Object

Declare:

> f : function_type;

so that *f*, when appropriately initialised, can be used (e.g. applied) as a function.

Function Instance Expression

The expression

> **function** : function_type [formal_part] function_body

makes an instance of the *function_type* with the indicated *function_body* which, depending upon the *result_type* for *function_type*, can be another function instance. If present, the *formal_part* allows renaming (but not retyping) of formal parameters. The conventional

> **function** name [formal_part] **return** result_type function_body

is retained, in effect as shorthand for

> **type** function_type [formal_part] **return** result_type;
>
> name : function_type := **function** : function_type function_body

Other abbreviated forms suggest themselves and are incorporated - see examples below.

Function Call

The expression

> f [actual_part]

as usual applies *f* to the *actual_part*. Because expressions may be function-valued, we can in fact have

> f [actual_part] ... [actual_part]

Lazy Evaluation

Function-valued functions provide the necessary mechanism for implementing "call by need", i.e., "call by need" actual parameters are transformed parameterless functions with the original actual parameter expression as body, and references to the corresponding formal parameters to calls on these functions. The refinements of "lazy evaluation" that avoid multiple actual parameter (re-) evaluations, one for each formal parameter reference, are achieved by introducing caching into the preprocessing scheme.

The mechanism is extended beyond the parameter passing mechanism so that any object may be declared as having a **lazy** type, so that assignments of values to it will only be evaluated on demand.

The requirement that the DAD programmer indicate explicitly when laziness is to be availed of is due to the hybrid nature of the intermediate levels of the hierarchy: at a level at which destructive assignment is still available, indiscriminate lazy evaluation of expressions is a recipe for confusion on a grand scale.

Streams and IO

A stream [12] is basically a list, of which usually both the head (element) and tail (sub-list) are evaluated lazily. The latter property in particular admits the processing of conceptually-infinite lists. The DAD type definition

```
    type S is stream of T;
```

is simply an abbreviation for

```
    type S is lazy record
        element: T;
        next : S;
    end;
```

Expressions generating streams S will evaluate only when the "next" field is referenced. A stream interface to input routines is straightforward, with a complementary presentation of output made available.

Referential Transparency

We have emphasised so far the positive aspects of functional programming that make it such a powerful tool. This power allows the application of restrictions on certain forms of expressiveness in functional languages in order to achieve mathematical simplicity but without detracting from their overall usefulness. These restrictions involve the elimination of the assignment concept from the language, and are consequently trivially implemented.

More positively, we are required now to replace "control flow" as expressed at statement level with expresion-level counterparts. Recursive functions adequately replace loops, but conditional- and case-expressions are necessarily introduced. Also, because destructive assignments are abolished, all expresion evaluation may be made lazy.

Parallelism

Finally, because DAD is defined in terms of Ada, DAD programs can inherit Ada's parallel facilities. While using tasking, for example, is not compatible with pure functional code, parallel versions of logical operations that extend their behaviour to the limits of computability can be defined and exported to the pure topmost level of the DAD hierarchy. The importance and usefulness of such facilities is detailed elsewhere [13].

DAD IMPLEMENTATION

Status

"DAda" - an early version of DAD that eschews first-class functions but instead provides just one of their consequences - lazy streams - has been extensively implemented to the extent of demonstrating viability (overloading etc. is not handled). DAD as defined above has been implemented up to the level of recreating streams and laziness from higher-order functions, and at the time of writing awaits re-integration with the remaining higher levels of the hierarchy (referentially-transparent state-free programming, parallel logical connectives).

Performance

Performance of higher-order functions leaves much to be desired, in view of their inheritance of the notorious penalties of tasking. The fact that dynamic task creation is essential to our solution leaves little room to hope that optimal implementations of special patterns of tasking will be of much use. The pragmatic solution available to us is to implement the consequences of first-class functions as special cases. For example, DAD's predecessor, DAda, provides a dedicated implementation just of lazy streams, with acceptable performance. Similarly, DAD itself actually implements lazy types through a purpose-built interface [14] to tasking that generates fewer tasks than required by the canonical derivation from first-class functions, still with acceptable performance. The problem remains of integrating these different base languages into a common family of successors in a unified Declarative Ada family tree.

The Future of Tasking

On a more philosophical note, there is a remarkable parallel between the status of Ada tasks today and that of procedures in PL/I some twenty-plus years ago - they both represent good ideas in program decomposition but are/were too expensive to use. Now that there are no longer credible arguments against the use of procedures on efficiency grounds, there can be no credible case against their use (save for some critical applications where dynamic storage management can't be trusted). Our hope is that our demonstration of the further expressive capabilities of tasking will help stimulate an approach to the solution of their implementation problems in much the same way as achieved for procedures long ago.

Design Refinements

As more examples are exposed below, it will become clear that the notation leaves room for improvement. DAD should at the present be regarded as an existence proof for the approach we are advocating, not as the limit of the technology. We welcome the reader's suggestions.

EXAMPLES AND APPLICATION

We briefly indicate how DAD is an adequate replacement for functional languages in the dimensions of expressiveness and of mathematical tractability.

DAD is Admirably Terse

Now, there are more sophisticated measures of a languages expressiveness than the bervity with which its programs may be written. Nevertheless, it is worth showing that one of the effective necessary conditions for legtimacy as a replacement functional language has been met.

The "Hamming numbers" problem involves generating a list of only those numbers whose prime factors are 2, 3, or 5 only. A Miranda-style definition of the required list "ham" looks like (full Miranda type information given for fairness of comparison):

```
ham :: [num]
ham = 1 : merge (mult 2 ham) (merge (mult 3 ham) (mult 5 ham))

mult :: num -> [num] -> [num]
mult n [ ] = [ ]
mult n (x : xs) = (n * x) : mult n xs

merge :: [num] -> [num] -> [num]
merge (x : xs) (y : ys)
      = x : merge xs (y : ys), if x < y
      = y : merge (x : xs) ys, if x > y
      = x : merge xs ys, if x = y
```

(Note the use of curried higher order functions for n-ary forms such as "merge": an application

```
merge Xs Ys
```

involves in detail first the application of "merge" to actual parameter "Xs" for formal parameter pattern "(x : xs)" with a resulting function of one formal parameter pattern "(y : ys)", followed by the application of that function to actual parameter "Ys".)

The DAD rendition assumes

```
package lists
      -- makes a type "list" with LISP operations "cons" etc.
      -- quite possibly an interface to streams
      ...
end lists;
use lists;
```

It proceeds:

```
type int_list_list is function (integer) return list_list;
type list_list is function (list) return list;
type list_list_list is function (list) return list_list;

function mult : int_list_list (n : integer)
      function : list_list (xs : list) is
```

```
        begin
            if xs = nil then nil
            else cons (n * car (xs), mult (n) (cdr (xs)))
            end if
        end;

    function merge : list_list_list (xxs : list)
        function : list_list (yys : list) is
            x : integer := car (xxs);
            y : integer := car (yys);
            xs : list := cdr (xxs);
            ys : list := cdr (yys);
        begin
            if x < y then cons (x, merge (xs) (yys))
            elsif x > y then cons (y, merge (xxs) (ys))
            elsif x = y then cons (x, merge (xs) (ys))
            end if
        end;

    ham : list :=
        cons (1, merge (mult (2) (ham)) (merge (mult (3) (ham)) (mult (5) (ham)))
```

The greater length of the DAD can be attributed to the generally busier concrete syntax of Ada as inherited by us, particularly: the lack of a pattern-matching style of formal parameter identification; and the need to surround actual parameters (or lists) with additional parentheses.

On the other side, note how some semantic innovations contribute to our faithfulness to the functional style and our maintenance of overall comparability to the Miranda rendition: conditional expressions; and recursive lazy constants (''ham'').

If greater terseness is required, currying can be supplanted with conventional parameter lists, viz.

```
    function mult (n : integer, xs : list) return list is
    begin
        if xs = nil then nil
        else cons (n * car (xs), mult (n, cdr (xs)))
        end if
    end;

    function merge (xxs : list, yys : list) return list is
        x : integer := car (xxs);
        y : integer := car (yys);
        xs : list := cdr (xxs);
        ys : list := cdr (yys);
    begin
        if x < y then cons (x, merge (xs, yys))
        elsif x > y then cons (y, merge (xxs, ys))
        elsif x = y then cons (x, merge (xs, ys))
        end if
```

```
end;

ham : list := cons (1, merge (mult (2, ham), merge (mult (3, ham), mult (5, ham))))
```

As well as shorter function headings, explicit function type declarations are omitted.

Formal Methods

A classic example is the demonstration that the function "len" to calculate the length of a list distributes over "append" which concatenates two lists. In a "genuine" functional language (Miranda-style) the definitions would appear as pattern-matching equations (equation numbers for reference):

```
(1)   len [ ] = 0
(2)   len (x : xs) = 1 + len xs

(3)   append [ ] ys = ys
(4)   append (x : xs) ys = x : append xs ys
```

The proof that

len (append Xs Ys) = len Xs + len Ys

proceeds by induction over "Xs" as a sequence of transformations following the equations:

Case Xs = nil

```
len (append [ ] Ys)
= len Ys                (3)
= 0 + len Ys            (arithmetic)
= len [ ] + len Ys      (1)
```

Case Xs = x : xs

```
len (append (x : xs) Ys)
= len (x : append xs Ys)      (4)
= 1 + len (append xs Ys)      (2)
= 1 + len xs + len Ys         (Inductive Hypothesis)
= len (x : xs) + len Ys       (2)
```

Note the intimate connection between the proof and the style of function definitions by recursion equations.

Formal Methods and DAD

Such equational notation is not available in DAD on the principle of retaining an obvious Ada style. However, from DAD function definitions, equational-style axioms defining function behaviours can be derived, allowing formal manipulations in the standard transformational style. This permits the employment of DAD in conjunction with the best of available supporting texts [10].

The precisely corresponding DAD function definitions are (assuming definitions of types inherited from previous example):

```
use lists;

type list_int  is function (l : list) return integer;

function len : list_int (xs: list) is
        begin
                if xs = nil then 0
                else 1 + len (cdr (xs))
                end if
        end;

function append : list_list_list (xs : list)
        function : list_list (ys : list) is
        begin
                if xs = nil then ys
                else cons (car (xs), append (cdr (xs), ys))
                end if
        end;
```

By inspection, the Miranda-style equations are easily recoverable from the DAD function definitions. However, the important criterion for the further development and refinement of our notation is recognised as the achievement of a closer approach to the equational ideal whilst maintaining an Ada flavour.

CONCLUSIONS

The following points are established.

- Software Development is a form of Engineering.
- Engineering Education relies on sound foundational material.
- Functional Programming provides the foundation for Software Engineering.
- Access to Functional Programming is enhanced through its insertion into existing language cultures.
- Ada represents a language culture of increasing significance in education and practice.
- Ada-flavoured Functional Programming is best achieved through a preprocessor-implemented dialect
- DAD captures the Functional paradigm adequately.
- Ada tasks need better implementation.
- DAD notation requires continuing refinement.
- The distance between Ada-flavoured Functional programming in DAD, and the ideal as represented by Miranda etc., that might appear to detract from DAD's pedgaogical usefulness, is easily bridged.

REFERENCES

[1] Bailes, P.A., Johnston, D.B., Salzman, E.J. and Wang, L., "DAda - an Ada Preprocessor for Functional Programming", Proceedings ACM 1990 TRI-Ada Conference, pp. 114-123, Baltimore (1990).

[2] Lloyd, B.E., "Professional Engineering in Australia, Antecedents and Futures", Australasian Engineering Education Conference, Preprints of Papers, Brisbane (1980).

[3] Neuman, P.G., "Risks to the Public", ACM SIGSOFT Software Engineering Notes, vol. 10, no. 5 (1985).

[4] Budde, R., Kuhlenkampe, K., Mathussen, L. and Zullighoven, H. (eds.), "Approaches to Prototyping", Springer, Berlin (1984).

[5] Henderson, P., "Functional Programming, Formal Specification and Rapid Prototyping", IEEE Transactions on Software Engineering, vol. SE-12, no. 2 (1986).

[6] Turner, D.A., "Functional programs as executable specifications", in Hoare, C.A.R. and Shepherdson, J.C. (eds.), Mathematical Logic and Programming Languages, pp. 29-54, Prentice-Hall, London (1985).

[7] Halpern, J.Y. and Wimmers, E.L., "Full Abstraction and Expressive Completeness for FP", Proc. Symposium on Logic in Computer Science, pp. 257-271, IEEE (1987).

[8] Stoy, J.E., "Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory", M.I.T. Press, Cambridge (1977).

[9] Abelson, H., Sussman, G.J. and Sussman, J., "Structure and Interpretation of Computer Programs", M.I.T. Press, Cambridge (1985).

[10] Bird, R. and Wadler, P., "Introduction to Functional Programming", Prentice-Hall International, London (1988).

[11] DARPA, "Curriculum Development in Software Engineering & Ada" (1991).

[12] Landin, P.J., "A Correspondence Between ALGOL 60 and Church's Lambda-Notation: Part I", CACM, vol. 8, no. 2, pp. 89-101 (1965).

[13] Bailes, P.A., Gong, M. and Moran, A., "An Expressively-Complete Functional Language", Technical Report 185, Department of Computer Science, University of Queensland, Brisbane (1991).

[14] Bailes, P.A., Johnston, D. and Salzman E., "A General Lazy Data Structure Extension for Ada", Proceedings of the 1991 Ada in Aerospace Conference (to appear), Rome (1991).

This Page Intentionally Left Blank

A Tool Supporting Programming in the Large for the Introductory Software Development Courses

Jaime Niño
and
Howard Evans
Department of Computer Science
University of New Orleans
New Orleans, La 70148
e-mail: jncs@uno.edu, hecs@uno.edu

Abstract

There is a conspicuous need to revise the teaching methodology used in introductory programming courses to provide students with a foundation for appreciation of the issues involved in the development of modern software systems. An important teaching component in software development must be software composition. In large measure, students are introduced to top-down development as a tool for system decomposition; software composition is limited to the components they developed themselves, along with the standard I/O packages.The objectives of this paper are two-fold; first to show that the rich set of program decomposition/composition tools provided by Ada gives the educator a great opportunity to raise the level of program development framework from statements and control structures to library units via reusability. The second is to introduce a software tool that allows the instructor to provide a environment for the use of these Ada characteristics as early as possible in the curriculum.

Introduction

There is an urgency to produce software engineers who have the methodologies and the skills to help industry address the software bottle-neck created by the high demand of software systems. The programming language Ada, as well as other modular oriented languages, have provided industry and education with tools to develop and teach these kinds of methodologies. Object orientation in particular and software reusability in general are methodologies which look promising in this problem.

In general, one important tactic towards improving software development productivity and quality is via software reusability. It has been recognized for quite a long time that one of the fundamental causes for the software bottleneck is the fact that new software systems are developed from scratch, for the most part. This is clearly a wasteful use of resources. Studies have shown that much of the code in a given software system can be found in other systems [1,2,3]. Other studies have been made to measure reuse of software; one of them [9] indicates that less than 15 percent of new code is needed in a new system. A lot has been written [1,2,3,15] on the possible benefits of software reuse; among the benefits are the need not to re-invent the wheel, and the opportunity to develop new software based on software that has already been shown to be correct. However, while software reuse is an approach that promises great payoff in the solution of the software crisis, it is an approach that remains unexploited. Reasons that have been given for the lack to fully explore this alternative range from the psychological, (the non-invented here syndrome), to the technical and the economical [3].

On the other hand, software reusability demands appropriate software tools, programming languages and software development methodologies to exploit this technique. The main issue to discuss in this paper is the foundation that is given to beginner programmers so that they will see the benefit and make use of software reusability; and than in particular, we expect they will recognize program development as an activity which does not start with an empty screen from which to build.

Program Development Framework

In large measure, students are introduced to top-down development as a tool for system decomposition; software composition is limited to the components they developed themselves, along with the standard I/O packages. At the same time students usually develop complete programs, which for the most part and due to time constraints, are small and very simplistic; these programs have no resemblance in any form or shape to the software that is available to them in PC's and in workstations.

Another characteristic of programs that students develop is that these programs are completely understood by them, due to their size. Thus, students develop a programming metal framework with the following characteristics:

- programs are started from scratch
- programs are developed individually; one persons's effort
- program's implementation details are well understood
- program complexity is at the reach of the programmer
- programs are static from the point of view of development
- code development is for complete programs
- program methodology: keep trying until it works

This software development framework becomes the standard by which they will routinely use with success; it is only until students are confronted with large and complex enough systems when this framework falls apart; they are completely unaware of this fact, and will continue trying what has proven successful on their eyes.

One important teaching component in software development must be software composition. Ada facilitates software composition using functions, procedures and packages via separate compilation. These library units can be used for development of programs by components, for team programming and for program reusability among others.

The task of an educator using Ada, is to provide an environment to instill in the student the use of these facilities as part of the development process; this provides the educator with an opportunity to raise the software development *abstraction level* of the student. In most instances these experiences are given to the students in a junior-level course in software engineering for instance, or via courses where projects of a rather large size are required. We claim that at this stage the educator is confronted with an existing programming methodology picked up by the student in the early programming courses, where program development is reduced to *statement-at-a-time* approach, and with the development characteristics as described above. The instructor must then set out to undo this mind-set in particular; the instructor must break the news to them that software development is not as straightforward as they have experienced; the instructor must argue against the pragmatic methodology that has proven successful to them until then; in general the instructor must set put to enlarge the abstraction scope to allow for the introduction of the problems, and the solutions, presented by the development of large and evolving software systems.

One important experience that a student should be exposed to early in the curriculum is team programming in particular, and of programming in the large in general. It is also fruitful to provide students with interesting problems whose team-based solution allows for the development of solutions that are closer in feel and flavor to software they routinely use in PC's or other environments. With the educational environment constraints, which among other include time and curriculum material it is rather difficult to provide the student with these experiences as early as possible in the curriculum.

From the teaching of software development view point, we feel very strongly about the need to instill in the student the reuse of library units and team work. Moreover, these two activities should be the core of software development education. We must produce graduates whose software development framework naturally encompasses these two activities. The goal for the use of these Ada facilities is to train students to enlarge their point of view when developing software. We must drive home the message that software development, for the most part, does not (should not) start from scratch, that is not an activity to be carried out within a software vacuum, as a solitary activity. These efforts are expected to raise their level of abstraction; a higher level of abstraction will motivate them to ask for and use sophisticated software development support tools effectively in their work-place; it will also give them opportunities to think and develop programs with a software library in mind, for reuse and for enlargement of it. In other words, they should think in terms of refining existing software or extending it, rather than always starting by building new software from the specifications. We must set the proper mental software habits as well as the proper software development framework in the early stages of the software development learning curve.

A methodology for teaching programming in the large

A goal of this methodology is to avoid the creation of the mental mindset of programming one-statement-at-a-time in an environment isolated from software and people as described earlier. The main underlying philosophy of this methodology is to teach the use and development of program (system) components within a programming team, and avoid the teaching of developing entire programs as the main goal.

In the introductory programming course using Ada, the student is initially given an overview of software development life cycle; software system structure is introduced from the implementation point of view using Ada; in particular, they are introduce to program structure using Ada; major components of this structure are: a driver (or main procedure), and library units such as subprograms and packages.

Students are initially introduced to functions and procedures as programming components; instead of being taught the writing of main programs with components. Using subprograms (functions and procedures), in a spiral approach, students are introduced to control units and software decomposition. Their code writing is only for subprograms. They are introduced to read and use library units, such as package specifications. They use library units in their subprograms. They are taught top-down design which they can use in the development of the subprograms they must implement; they are also introduced to bottom-up design, to understand the use of the library units available.

The next step in the learning process is the writing of drivers (also called main procedures) which will strictly use library units; the design methodology exemplified here is bottom-up design, to understand the use of the library units available.

A last step in the learning process of this methodology, students are given complete programs to implement, where they must supply the driver and some program components; others will be available through the library. The design methodologies to use here is top-down and bottom-up. The goal here is to develop a design which will identify the components of the program (functions and procedures) and the identifications of those parts which will come from a library, and those which need to be developed.

The main gains of this methodology are:

- large abstraction scope, early in the learning process
- control of the methodology used to write programs
- an audience more prone to understand and appreciate software engineering problems and solutions
- stress in reading of specifications and code written by others
- satisfaction in writing parts of interesting and complex programs

Turnin facility

The turnin facility to be described below, can be used in a straightforward manner to allow for a control submission and testing of programming assignments. One of the points of this paper is to show how it can support the teaching methodology as described above; in particular, a methodology of programming where the student does not develop complete running programs from the start, but only component of a program. The rest of the program is provided via library units provided by the instructor.

In essence *turnin* is a facility which allows for a control submission, testing of students programs, and output analysis.

By control submission we mean the activity which allows

- the description of thc software component which will produce a running program
- the source of each of the components
- the hierarchical organization of the components

By testing we mean the, possibly multiple, execution of the resulting program after the controlled submission phase using hidden data.

By output analysis we mean the automated examination of the program's output.

Specifically, the Turnin Facility consists of several command procedures that perform the following tasks:

- Helping instructors set up the facility for a specific assignment (Turnin_Setup, Turnin_show commands)
- Assisting students in turning in their programs (turnin command)
- Testing student programs according to the instructor's specifications (turnin command)

- Keeping track of the students that have turned in (turnin_count command)
- Cleaning up after an assignment is completed (turnin_cancel, Turnin_delmail commands)

What follows now is a description of each of the major commands available through the Turnin facility.

Turnin_setup Command

This is an interactive process where the instructor specifies the following information:

- Assignment name: it will be used by students to invoke turnin.
- Language: source code language.
- Due date: date and time when program is due.
- Compilation qualifiers: specification of appropriate compilation switches.
- the list of modules to be compiled during the turnin: specification of the number of modules to be submitted by a student, and of those submitted by other sources. Also, a specification of compilation order, to indicate the program's driver.
- link qualifiers: specification of appropriate link switches
- data files: specification of the data files to be used per run.
- files echoed: specification whether data files used for testing will be echoed.
- command file before compilation: specification of a command file to be executed before compilation.
- .command file after compilation: specification of a command file to be executed after compilation.
- command file after each execution: specification of a command file to be executed after each execution.

It is possible to provide a command file to be executed before compilation. It will receive one parameter which is a copy of the compilation list after filling in all of the student module names. This command file can do what it wants with this list of files. The use of it is to complete source files submitted by students, as a some sort of implicit include, by inserting code in source files in the appropriate places. The command file must invoke a program that will do the desired source change.

It is also possible to provide a command file to be executed after the compilation step in the turnin process. After all of the modules involved in the turn in are compiled (student and instructor), this command file will be called. It will receive one parameter in P1 which is a copy of the compilation list after filling in all of the student module names. This command file can do what it wants with this list. We have used this capability to do a DIFFERENCES between the student's source file and the original.

It is also possible to provide a command file to be executed after each data set is run. This command file will receive three parameters in P1, P2, and P3. P1 will be the actual file specification for the output file which was just created by the student's program. P2 will be the data set number (1, 2, 3, ...) of this run. P3 will be the status value returned by executing the student's program; it can be used to determine whether the student's program even worked. If you want to run a program which will read and verify the contents of the output file (automatic output check!) the command file must invoke a program (supplied by instructor) to do the output analysis.

Turnin_show Command

This command display the turnin_setup characteristics of a given assignment.

Turnin Command

This command will be used by the students to submit their program assignments. The assignment name must be specified. This process is interactive where by the students will be prompted for the name of all the modules which they must submit to form a complete compilation unit. If the files specified in this step exist the turnin command proceeds to compile, link and execute the students program. Failure in compilation will terminate the process. Every time that a student issues a Turnin command and specifies successfully the files to be used, the instructor will receive a mail message indicating the student name, the assignment name and the time when it was issued. At the end of the turnin process, the students is issue a hard copy of the source listing of the program and the output generated. This file is handed out to the instructor by the student.

Turnin_count Command

This command checks and produce a list of the students who submitted the assignment, the time submitted and the number of times the made a submission of the assignment.

Turnin_cancel Command

This will erase the setup information on that specific assignment making students unable to turn in any more programs.

Turnin_delmail Command

When you have finished with a specific assignment you may clean up your mail from this assignment's turnin messages. As a security precaution, this command will delete only the turnin messages found in the Mail folder TURNIN_MESSAGES. Notice that turnin messages end up in that folder after they have been counted with the TURNIN_COUNT command.

Currently, the Turnin Facility supports several of the VAX-11 compiled or assembled languages. This list includes Ada, C, COBOL, FORTRAN, MACRO, Pascal, Icon, LiSP. The reason for this list, is based on the fact that these languages are used in different computer science courses for programming assignments.

This software tool as it currently exists, is implemented to run in a VAX/VMS environment. No ports to other environment has been done yet. We have been using this facility for many years. It have proven very effective for assig ment submission and testing. We expect to have an implementation of this kind of facility for a Unix environment by the end of the fall of 91.

Turnin facility as an Environment for Programming in the Large

The Turnin facility directly supports the methodology as described above. Students submit parts to form a working program and the facility puts the program together to form a complete running program.

In particular, the turnin can be setup to:

- accept one or more library units from the student (functions, procedures, or main procedure)
- provide all others library units (packages, subprograms, main procedure) to form a running program.
- hide language features that prove difficult to introduce at a given level (such as instantiation of generics, declaration of user defined types)
- test program, and specifically user provided units.
- generate an annotated analysis of output generated by program.

Using this Turnin facility, the instructors can place the students in a team environment, where the instructor and assistants form one team and each student or group of students form other teams. The main goal of this approach is to introduce programming from early in the learning cycle as a complex activity which requires many people to develop interesting programs and software systems.

Future work

Currently the authors are in the process of implementing a Software Development laboratory around a Sun server and 25 Sun stations, supplied with windowing environment and case tools to continue our effort of increasing the programming scope and abstraction level as early as possible in the learning process. We are currently developing a curriculum based on a spiral introduction of tools and techniques used to develop modern-day software; this curriculum is to be implemented in the three core programming courses our students must take. These courses will be four-credit courses, where one hour per week will be dedicated to supersived laboratory instruction. In this laboratory students will be introduced to use of tools to support programming as a team activity . In particular we are redesigning the Turnin facility to provide students with a control testing facility of their software with data provided by the instructors, producing annotated output where needed.

Conclusion

The programming methodology currently used to introduced programming as an activity, needs to be revised and must to be supported with proper software tools to be effective. Students that have finished two semesters of programming classes acquire a very distorted view of program development. This view must subsequently be fought against when students are introduced to the issues of developing large, complex and evolving programming system. The methodology presented and supported by the Turnin facility, we believe, is an step in the right direction; software systems should not be created from scratch, in a software vacuum and a solitary activity; we must not use and encourage teaching methodologies which support this view.

Bibliography

[1]•Biggerstaff, Ted, Perlis, Alan, eds. Special issue of IEEE Transactions on Software Engineering on software reusability, SE-10(5) 1984.

[2]•Biggerstaff, Ted, Perlis Alan, eds. Software Reusability. Vols I, II. ACM Press. Addison-Wesley Pub. Co. 1989.

[3]•Biggerstaff, Ted, Richer Charles. Reusability Framework, Assessment, and Directions. in [2]

[4]•Dijkstra, E. W. On the Cruelty of Really Teaching Computer Science. A debate on Teaching Computer Science. CACM, Vol 32, Num 12. Dec, 1989, pp1398-1404.

[5]•Jones, T.C. Reusability in programming : a survey of the state of the art. Special issue of IEEE Transactions on Software Engineering on software reusability, SE-10(5) 1984.

[6]•Meyer, B. Reusability : The case for Object Oriented Design. in [2] Vol II. pp. 1-34

[7]•J. Niño. "Object Oriented Models for Software Reuse". Proceedings of the IEEE Southeastcom. March 1990.

[8]•J.Niño. "A Design Methodology for Object-based Languages". Proceedings of the Fifth Annual of the Ada Software Engineering Education and Training Team (ASEET) Symposium. August, 1990

A Top-Down Toolbox Approach to Teaching the Ada Programming Language

Thomas B. Hilburn and Iraj Hirmanpour
Department of Aviation Computer Science
Embry-Riddle Aeronautical University
Daytona Beach, FL 32114

ABSTRACT

This paper discusses a multi-year project at Embry-Riddle Aeronautical University that is using a new instructional strategy to develop three introductory courses: Computer Science I and Computer Science II, and an Algorithms and Data Structures course. The instructional strategy concentrates on use of Ada and a top-down approach to teaching introductory computer science courses that is centered around realistic, moderately complex software systems that would be of interest to a freshman computer science student. With this approach, students are provided with a toolbox that will contain a set of software tools that will be used in implementing and modifying parts of the software system. In the first course students view the software system and the software tools as "black boxes" and concentrate on how to use the black boxes. In the second course students look inside the boxes and concentrate on how to implement the lower level details of the software system and the toolbox. In the third course the students analyze and enhance the data structures and algorithms used in the first two courses. The project is in its first year and it is expected to take at least three years to complete.

INTRODUCTION

This paper describes a new approach to teaching introductory computer science courses. The purpose of the new approach is to show at an early stage (in the first course) the relevancy of concepts taught in the computer science curriculum. Students at a very early stage are introduced to a complete computer software system. The student is also provided with a set of software tools that will allow the student to implement and modify parts of the software system. The student will initially view the software system and the set of tools as "black boxes" that can be conveniently manipulated to produce meaningful results. As students progress through the introductory courses they will be exposed to increasing levels of detail about the makeup of the "black boxes". Although aspects of this approach have been used in introductory courses (see [Pattis,1990]), the authors argue that such an approach should be a the center of the lower level programming courses. This "top-down" approach to teaching computer science should result in increasing students' interest in the subject, and it is hypothesized that this heightened interest will contribute to the reversal of the decline in freshman interest in computer science.

This "top-down toolbox Ada based approach" is being implemented by the Aviation Computer Science Department at Embry-Riddle Aeronautical University (ERAU) over a three-year time interval. The project involves the development of three courses: Computer Science I, Computer Science II and an Algorithms and Data Structures course. This paper is an interim report on the progress of the project and is intended to promote discussion of the merits of such an approach.

PROBLEM STATEMENT

The authors believe one of the reasons for lack of interest in science and engineering lies in the fact that the science curriculum is often presented in such a highly theoretical and abstract way that the contemporary student fails to see the relevance to real life problem-solving and, therefore, loses interest and commitment. Science and engineering education traditionally has made the assumption that students need to obtain a large body of basic knowledge and fundamental concepts before the relevancy of the knowledge and concepts can be brought into the curriculum. Most curricula are based on this assumption and students do not experience real life problem-solving until they are juniors or seniors in their field of study.

The traditional approach is therefore "bottom-up" and assumes that students must master details of many individual concepts before their relationship to the solution of real-world problems can be shown. Typically, it is not until the latter part of the junior or during the senior year that students begin to see how the individual concepts they have been learning can be synthesized and incorporated into a solution of a problem that is both interesting and realistic. In the first two years of a typical computer science curriculum, students are taught how to write code in some modern high-level language and introduced to the concepts of structured programming, design and analysis of algorithms, data structures, and computer organization. The examples used to illustrate these concepts are usually artificial and often fail to show real-life relevancy.

This bottom-up approach has two disadvantages:

(1) it fails to show relevancy of concepts at an early stage, causing some students to lose interest in the subject;
(2) those who succeed leave the introductory courses with the notion that attention to detail is the first step and the most important part of problem solving.

In an upper division software design course the paradigm is reversed and conceptualization and organization of the problem solution is emphasized as paramount to the problem-solving activity and students are taught to postpone details. This mid-course reversal of instructional strategy confuses students and many graduate without a full understanding of the problem-solving approach that higher level courses stress.

A similar paradigm is followed in teaching Aerospace Engineering, an engineering program offered at ERAU. In order to design an airplane students need to know concepts from aerodynamics, fluid mechanics, thermodynamics and aircraft structures. These concepts are dependent on subjects such as physics and mathematics. It takes a student three years of concept study before he/she is faced with the central problems and issues of aircraft design in an aircraft design course.

Students are more apt to enroll and stay with science and engineering programs that show relevancy early. The Aeronautical Science (flight) program at ERAU is such a program. Aeronautical Science students can begin flight training during their freshman year; as they progress through their curriculum they can see the relevancy of such courses as mathematics, physics, aerodynamics, and meteorology. There are other examples of such programs and courses. For example, in an introductory accounting course students solve realistic problems. Computer Science, on the other hand (along with many other science and engineering programs), takes a longer time to show relevancy and it is believed that this is one of the factors that contributes to the higher attrition rates and smaller enrollments in the program.

PROPOSED SOLUTION

General Approach

Rapid advances in technology during the past three decades have had a profound impact on lifestyles and attitudes. Improvements in transportation and communication brought on by computers, telecommunication and aviation has accelerated the exchange of information and generated new knowledge and ideas at an explosive rate. Young people see their learning experience in this context and, as a result, they expect (and even demand) relevancy in their educational activities. We believe that new instructional strategies that have as one of their goals showing relevancy at the introductory level will attract and retain more students in the discipline.

Instructional Strategy

To test this hypothesis, it is proposed to change the instructional strategies and materials in the introductory computer science courses. Rather than use the traditional bottom-up method of instruction, we propose a top-down approach that first exposes the student to a realistic, moderately complex software system (or several such systems) that would be of interest to a freshman computer science student. In addition, the student would be provided with a toolbox that would contain a set of software tools that would be used in implementing and modifying parts of the software system. The toolbox would include such things as the following: a set of generic data structures (sets, ordered lists, stacks, queues, trees, and graphs); a package of graphic tools; a package of math functions; a package of string operations; and a set of commonly used utilities such as input/output, and sorting and searching. Initially, the details of the implementation of the software system and the set of tools would be hidden from the student and he/she would view them simply as "black boxes" that yield output when specific inputs are provided. In fact, in the first course the student would not see any of the details of what is "inside" a black box. Rather the student would learn about the functionality of the black boxes and how to use them. For example, in Computer Science I, students might be given an assignment to set up a scheduling system for student pilots that requires queuing a set of student records. The student would have to understand what a "queue" abstract data type is and how to use the "queue" black box provided in the toolbox. However, the student would not have to know or understand, at this point, how the queue is implemented (arrays or pointers, linear or circular, etc.). In essence, we propose that a "fourth-generation" type programming environment be used in teaching Computer Science I.

In Computer Science II, the student would continue with the study of one or more complete software systems. The student would be exposed to lower levels of detail about the system and would look inside the black boxes that were used in Computer Science I. The emphasis in this second course would be on implementing the data structures and algorithms used in the first course. For example, students would implement the queue tool used in an assignment in Computer Science I.

Following Computer Science II, students would take a course in which more advanced features of data structures and algorithms are studied and in which there is an emphasis on analysis and comparison. Hence, the student first learns to "use", then learns to "implement", and finally learns to "analyze".

Instructional Software Considerations

We are currently developing software systems that address several different aviation/aerospace problems that are appropriate for use in our introductory computer science courses. One, for which we have built an prototype version, is an "Automated Flight Planning/Scheduling System" (AFPS) that can be used by a student pilot to write a flight plan and schedule a flight. Three other systems being developed are a "Spaceship Docking Simulation" (SDS) that simulates simulate docking of a spaceship and an orbiting lab (see [Pooch,1989]), a "Aircraft Detection System" (ADS) that detects distances and bearings between aircraft in an airspace (see [Hilburn,1991]), and an "Operational Flight Control System" (OFCS) that will simulate an aircraft avionics system that provides aircraft control functions and displays aircraft status information. We are currently using the Ada programming language in our introductory course and, hence, all systems will be developed in Ada. We believe Ada best captures the spirit of the proposed instructional strategy because its "package" concept supports the abstraction and information hiding that is a crucial element of our approach. An Ada package specification for a tool (or group of tools) would provide the student with a "black box" view of a tool.

In addition, we are in the process of developing a toolbox that supports the course objectives. Although there are some off-the-shelf tools available, the proposed instructional strategy dictates that we develop the toolbox in-house. First, since the tools are not just to be used for software development but will be used by the instructor as a tool to teach computer science concepts, there are special design considerations. (e.g. there are pedagogical considerations in Ada coding related to context clauses, private types, generic program units, input/output and exception-handling.) Second, since these tools will be used as examples to illustrate programming principles it is essential that we have the source code for all the tools.

Course Content

An important part of our project involves the development of a comprehensive course outline, with supporting material, for each of the introductory courses. The outlines will provide topics to be covered along with a list of objectives for each topic, suggested activities and assignments, and a description of supporting material to be used. A great deal of this work has been completed for Computer Science I and less detailed outlines have completed for the other courses. The Appendix contains a descriptions for some of the topics for Computer Science I. It gives the flavor of the instructional strategy and illustrates some of the materials and activities to be used in the top-down black box approach:

a. In Topic 1, Introduction to Software Systems, the students get a look at a complete system definition and a user's manual, and get to try out some test data on a system of moderate complexity - a set of activities not usually included anywhere in the first course.
b. In Topic 2, Top-Level Design of a Software System, the student gets to do some meaningful documentation. For example, the student might rewrite some portion of the user's manual that he/she found unclear or incomplete, or the student might add some documentation about the purpose of a program module. In the traditional introductory computer science course, a student's first

experience with documentation typically involves a fifteen line program that only involves text input/output - the student really cannot see the need for such work and resents having to do it.

c. In Topics 4 and 5, Introduction to Subprograms and Control Structures, the student will be involved in developing and modifying a moderately complex system using modules already developed. In one sense, the student will be applying the same concepts (control structures and subprograms) as in a traditional course, but he/she will be using them in a context that is more interesting and relevant, and that emphasizes software engineering principles associated with the development of large software systems.

IMPLEMENTATION ENVIRONMENT

The proposed concept is a natural outgrowth of ongoing pursuits of ERAU's Aviation Computer Science Department. An "aviation computer science" curriculum has been developed at ERAU that follows the Association for Computing Machinery (ACM) and the Computer Science Accreditation Board (CSAB) curriculum guidelines for undergraduate computer science. The program integrates computer science with an aviation applications focus. The program has several innovative (perhaps unique) features. Two features germane to this discussion are: integrate software engineering principles throughout the curriculum including the first programming course, and show relevancy of computer science to the solution of aviation/aerospace problems.

In support of software engineering integration, the department has adopted Ada as the core programming language in all software development courses, including the introductory programming courses. In addition, the department has developed a "Software Development Manual" that describes a software methodology that will be used in all courses involving software development. Students are exposed to elements of the methodology (along with other formalities of software engineering) in the introductory computer science courses. Students develop and implement their programs in Ada running on a network of Sun Workstations running in an X-window environment. Although the toolbox is not complete we have developed initial versions of the following: a set of generic data structures; a package of graphic tools; a package of math functions; a package of string operations; and an input/output package.

In support of aviation applications, the department has a unique resource: the Airway Science Simulation Laboratory (ASSL). The ASSL is staffed jointly by faculty from the Computer Science Department and the Aeronautical Science Department. The laboratory includes the elements of the National Airspace System (air traffic control simulators, pilot training simulators, meteorology laboratories, etc.) in an interactive, intelligent simulation training configuration. Advanced students are afforded the opportunity of working with faculty on aviation projects. Lab facilities and research projects provide real-world practical experiences to our students that are not normally found in an undergraduate program.

FIRST YEAR EXPERIENCES

Since 1987 the Algorithms and Data Structures course at ERAU has been taught using the Ada programming Language. In 1989 Ada became the primary language in the computer science curriculum and it was introduced in the first course, Computer Science I. In the first year it was taught in the typical "bottom-up" fashion. In the 1990-1991 academic year the "top-down" approach discussed in this paper was used in teaching Computer Science I and Computer Science II to approximately 100 students. Although the software and other supporting material was not complete, the course objectives and instructional strategy were based upon the concepts we have been discussing. Our software system examples and toolbox components consisted of a combination of commercial products and locally prepared products. The textbook used, [Volper,1990], was selected because, of the available texts, it most closely followed the course philosophy. The student laboratory used a Meridian AdaVantage compiler running on Sun Workstations and a PC version was used for classroom demonstrations. Although the instructors had some initial problems with application of the new approach (mainly due to the problems with instructional support material), their overall opinion of the results was very positive. The students taking the courses gave generally high marks to the "top-down toolbox" approach; however, in many cases they did not have anything with which to compare it.

It is too early to make any definitive statements about the viability of our new approach, but informal assessments of students completing the "top-down" Computer Science I compared to the previous "bottom-up" Computer Science I (using Ada and Pascal) show improvements in attitudes and capabilities related to the principles of software design and development. As additional instructional support material is developed and as students move through the curriculum into their upper division computer science courses, a more realistic and meaningful evaluation can be carried out.

CONCLUSIONS

We expect that our proposed approach to teaching the introductory courses will increase student interest in the study of computer science and will decrease the attrition rate in these courses. In fact, we believe that the success of this approach will motivate new students to enter our aviation computer science program. We also feel that students who complete these courses will go on to their upper division courses with a better understanding and appreciation of the computer science discipline. It is expected that the aviation computer systems developed for the first two courses will be used in subsequent courses by exploring implementation details and modifying and enhancing the systems. As students learn concepts in the areas of data structures, graphics, artificial intelligence, and simulation, they will be able to apply them to the systems that they are already familiar with. These students will be able to take on more challenging and realistic projects in their junior and senior years and will graduate with better problem-solving skills. They will enter the job market with a better appreciation for the complexities involved in the design and development of large scale software systems.

Finally, we believe this new approach will help the faculty become better teachers of computer science. The software systems and tools, and the outline with its supporting materials will provide a rich resource that will assist the instructor in organizing and presenting the course concepts. The aviation software systems will provide a focus for the introductory courses that is not available in current commercial courseware. The project materials will also support appropriate uniformity between various instructional approaches to the same concept and provide continuity between different concepts.

REFERENCES

Hilburn, T.B., "The Use of Ada in the Simulation of An Aircraft Detection System", **Proceedings of 1991 Conference on Simulation Technology**, Orlando,FL, October 1991.

Pooch, U.W. and Tanik, M.M., **An Ada Courseware**, Meridian Software Systems Inc.,1989.

Pattis, R.E., "A Philosophy and Example of CS-1 Programming Projects", **SIGCSE Bulletin**,Vol. 22,No. 1, February 1990.

Volper, D. and Katz, Martin D., **Introduction to Programming Using Ada**,Prentice-Hall,1990.

APPENDIX

COMPUTER SCIENCE I
(example Course Outline)

Topic 1: INTRODUCTION TO SOFTWARE SYSTEMS

objectives:
a. students will understand the basic organization of a digital computer, the difference between hardware and software, and the function of an operating system;
b. students will be able to use a moderate size realistic software system;
c. students will be able to understand the system definition statement and the user's manual for the software system under study;
d. students will understand the purpose of test data and will be able to create a set of the test data for the system under study.

class activities:
a. discuss the organization of hardware/software systems and the role of operating systems;
b. discuss a system definition statement and provide students with the system definition and user's manual for the system under study;
c. demonstrate the use of the system under study.

student activities:
a. read handout on introduction to computer systems;
b. construct several sets of test data and run each set on the system under study
(e.g. use the SRS to dock the spaceship with the lab for different sets of initial conditions.)

supporting material:
a. a handout on introduction to computer systems;
b. a system definition statement (problem definition, goals of the system ,capabilities and constraints, user characteristics, etc.);
c. a user's manual;
d. the software system (executable program)

Topic 2: TOP-LEVEL DESIGN OF A SOFTWARE SYSTEM

objectives:
a. the student will understand how the system definition statement was used to carry out a top-level design of the system under study;
b. the student will understand how a high-level language program is organized (program header, declarative part, executable part);
c. the student will understand the role of modules in the design

of software and know the meaning of "divide and conquer";
d. the student will understand the role of internal and external documentation of a software system;
e. the student will understand the purpose of a compiler and linker;
f. the student will be able to use a text editor;
g. the student will be able to modify an Ada source program, and then compile and link it.

class activities:
a. discuss top-level design of software systems and illustrate with the system definition statement and the Ada source code for the system main program;
b. discuss internal and external documentation and illustrate with the system user's manual and Ada source code for the system main program;
c. discuss the use of a text editor;
d. discuss the purpose of a compiler and linker and illustrate their use;

student activities:
a. read handouts on top-level design, the text editor, and the use of the Ada compiler system;
b. use the text editor to modify the internal documentation of the system main program;
(e.g. insert a description of each of the modules referenced in the SRS.)
c. compile, link and run the system.

supporting material:
a. handouts on top-level design, the text editor, and the use of the Ada compiler system;
b. a text editor;
c. the system definition statement and the Ada source code for the system main program.

Topic 3: INTRODUCTION TO ABSTRACT DATA TYPES

objectives:
a. the student will be able to define the term abstract data type (ADT);
b. the student will be familiar with the standard Ada data types;
c. the student will be familiar with literals, variable names and named constants;
d. the student will understand how to declare and initialize variables and declare subtypes and new data types;
e. the student will understand how to modify the system parameters and analyze the effects of such modification;
f. the student will understand how Ada packages can be used to declare data.

class activities: (to be completed during the project)

student activities:
a. read handouts on Ada data types and objects, and Ada packages;
b. modify the system parameters, recompile and link;
(e.g. in the SRS, one might modify the time increment used in the simulation or vary the mass of the spaceship.)
c. run the modified system and analyze the results.

supporting materials:
a. the Ada source code for the system package used for data declaration.
(the rest to be completed during the project)

Topic 4: INTRODUCTION TO SUBPROGRAMS

objectives:
a. the student will understand the subprogram concept;
b. the student will understand how subprograms are used to modularize a program;
c. the student will understand subprogram calls and parameter passing;
d. the student will understand Ada package specifications;
e. the student will be able to write a program to solve a problem using the modules of the system being studied.

class activities: (to be completed during the project)

student activities:
a. using the system definition statement and the specification of a system package of subprograms, the student will write a program that implements the system;
(e.g. a new system, say the AFPS, could be introduced at this point and the student could be asked to write the system main program.)

(the rest to be completed during the project)

supporting material:
(to be completed during the project)

Topic 5: CONTROL STRUCTURES

objectives:
a. the student will know the purpose and understand how to use the selection and iteration control structures;
(the rest to be completed during the project)

class activities:
(to be completed during the project)

student activities:
a. use control structures to extend the capabilities of the system;

(e.g. the AFPS could be modified to allow multiple flight plans to be created.)

b. use control structures to write the code for the subprograms in the systems under study and test them by "plugging" them into the appropriate system.

PRESENTATION

INTRODUCTION

ERAU

ACS DEPARTMENT

ACS CURRICULUM

Ada
Software Engineering Across Curriculum
Software development manual
Project courses

B. PROBLEM STATEMENT

Bottom-up Curricula
- Takes too long to show relevancy
- Fails to teach "big picture" thinking
- Requires paradigm reversal

PROPOSED SOLUTION

Top-Down Approach

- Captures student interest early
- Improves problem solving abilities
- Is highly resource intensive
- Ada is ideal for it.

- General Approach
- Instructional Strategy
- Instructional Software Considerations
- Course Content

PLAN TO INTEGRATE INTO THE CURRICULUM

ASSESSMENT AND EVALUATION OF OUTCOMES

J. BIOGRAPHICAL SKETCHES OF THE INVESTIGATORS

1.Dr. Thomas B. Hilburn, Professor of Computer Science, joined the ERAU faculty in 1973. Prior to that he served in the United States Navy from 1962 until 1969. While in the Navy, he taught courses in computer technology, inertial navigation and satellite navigation. He received his Ph.D. in Mathematics from Louisiana Tech University in 1973 and has completed graduate work in computer science and computer engineering at the University of Central Florida and Rochester Institute of Technology.

Since arriving at ERAU, Dr. Hilburn has taught in both the Aviation Computer Science and Mathematics Departments. He has received numerous awards for teaching and service to the University. Dr. Hilburn is currently active in both the Association for Computing Machinery and the Mathematical Association of America, and is a regular reviewer for Computing Reviews. He has taught the Ada programming language in various courses and seminars for the last four years. His current research interests include reusability and the use of Ada in representing discrete mathematical structures. He has delivered several papers in this area in recent years.

2.Dr. Iraj Hirmanpour, Professor of Computer Science, is the Chairman of the Aviation Computer Science Department. In addition to ERAU he has served on the faculties of Illinois State University and the University of North Carolina at Charlotte. He received an M.E. degree in Computer Science from the University of Florida in 1970 and an Ed.D. in Computer Science Education from Florida Atlantic University in 1980.

In addition to teaching, Dr. Hirmanpour has acted as a consultant on software development methodology to numerous commercial firms and government agencies - both in the U.S. and abroad. He has designed and presented workshops to professional groups and is a frequent speaker at local professional meetings. His interests include information system modeling and software engineering. He is a member of the ACM and IEEE's Software Engineering Group.

Using a Language Sensitive Editor and Ada in Computer Science I-II

Dr. Dennis S. Martin
Department of Computing Sciences
University of Scranton
Scranton, PA 18510-4664
MARTIN@JAGUAR.UOFS.EDU

Design, documentation, and development--there is an ever increasing gap between beginning programming skills and the skills required of a professional programmer. It is essential that this gap be narrowed and that a student start programming with careful consideration of the philosophy and principles of software engineering. In a sense, these concepts are "language-free" in that the major attention must be placed on concept development rather than language syntax. However, these concepts are really not understood unless the student implements problem solutions in a programming language, preferably a programming language that directly supports these concepts. Based on several years of experience, we have found the use of a Language Sensitive Editor (LSE) to be exceptionally effective at the CSI-CSII level to teach a language-independent approach to the concepts of programming. The language Ada is an ideal companion to the LSE, providing the direct language support needed.

A very important part of teaching concepts is to force the student to articulate the design of their algorithms before the code is written. The student must learn to develop useful though not necessarily extensive internal documentation for a program. For most students, documentation is, at best, an add-on after otherwise finishing the program. The usual reasons given for this behavior include not knowing what documentation is needed, not knowing where the documentation is appropriate, and not wanting to invest the amount of time necessary to type in the documentation. In the past, we have attempted to solve this problem with written documentation standards. The results were not impressive.

The key to good design is modularity including both procedural abstraction and data abstraction. Actions and objects must be accessible only through clearly defined interfaces which contain formal specifications of the action or object. (See, for example, Sommerville's text on Software Engineering.) Ada allows these interfaces or specifications to be separately compiled. Students can start by writing programs using already constructed units having access only to well-documented specifications, not to the body of the code. Later, students can learn design by having to write, document, and compile a mainline and specifications of its supporting units. We have found it desirable to have students hand in such specifications reasonably soon after an assignment is given but before the student has had a chance to fully write the program. These can be evaluated for correct design and are effective in verifying the high-level design before lower level coding is started.

FORTRAN allows separately compiled functions and procedures but uses only the body of code and does no type checking. Data abstraction is difficult. Modula-2 allows separate compilation units with definition and body (good object abstraction) but forces a separately compiled procedure or function to be hidden within a module confusing the ideas of data abstraction and procedural abstraction. Standard Pascal does not support separate compilation units so the suitability of a particular (non-standard) extension varies.

We are using the VAX Language Sensitive Editor on a VAX/VMS system. The LSE is user modifiable and we have chosen to customize it to reflect our view of appropriate documentation style. As an additional benefit, since the LSE uses Extended Backus-Naur Form (EBNF) as its paradigm, it enhances the use of EBNF in describing the syntax of a language.

At any time during development, a program source file contains both code (terminals) and placeholders (non-terminals). Placeholders have distinguishing delimiters, such as %{ }% or %[ ]%, which are standard printable characters but syntactically meaningless in the language. A new file contains only a root placeholder such as %{compilation unit}%. A placeholder is either a single element, such as an identifier, which must be typed over, or a language element which is expanded to produce a template of code and other placeholders. A placeholder is expanded by placing the cursor on it and typing a control character.

The fill-in-the-blank format of the LSE has solved most of the problems associated with documentation. When a student expands the template for a procedure, function, or package, it contains the names of the features that must be addressed. There is no question about what is required or where it should go. Students do less typing as the headings are already typed for them. Text automatically wraps to the next line, properly indented. Students are much less likely to hand in a program with insufficient or nonexistent documentation. Faculty are much less likely to accept such work.

At the simplest level, procedures and functions need a careful explanation of input, processing, and output--what information is available, what a program is supposed to accomplish, and the results desired. This pre-condition, post-condition, and functional description framework, with the input/output information supplied by formal parameters and non-local referencing, forms the specification of a function or procedure. It provides sufficient information needed for its proper use.

The story for data abstraction is less clear. Functional abstraction has a long history but understanding object-oriented approaches is a continuing development in the field. Proper documentation standards will also continue to develop. At this time, we require a general overview of the object and a careful description of what types, procedures, functions, and exceptions should be available. Judicious use of private and limited private types encapsulate the object itself properly. In CSI and CSII, it is very appropriate to insist on exceptions (and exception handling) and to add the concepts of generic objects and operations as part of the conceptual development. These topics would be almost impossible to treat at this level in any other language.

The appropriate style to comment a loop has caused us much discussion. Ideally, students should have a trail of assertions which lead from the pre-condition to the post-condition but this is very difficult for beginning students and is too laborious to be done in professional practice on a regular basis. We have compromised by requiring that all loops have a weak form of "loop invariant". While few students can give precise and correct invariants at this level, they can articulate what the loop should do and how it will be exited. This results in better designed and implemented loops. Since Ada allows exiting a loop in the middle, we consider the location of the loop invariant to be part of the loop design process.

With an LSE, our students can concentrate on concepts rather than syntax, writing better quality code with less need for extensive syntactic corrections. Our original concern that introducing this tool to beginning students might increase their confusion or might produce proficient tool-users lacking understanding has been found to be unwarranted. We believe that an LSE should be used as early as possible to enhance student learning of program design and documentation supporting good design

As computer professionals, we need to use more computer-intensive tools to enhance the learning of computer science. Good tools, used properly, can be effective to help students learn to be able to create good design and then to transform that into good code, free from both syntactic and logical mistakes. The emphasis in computer science must shift from learning syntax to software engineering.

References.

Sommerville, Ian, Software Engineering, third edition, Addision-Wesley, 1989.

Guide to VAX Language-Sensitive Editor and VAX Source Code Analyzer, Digital Electronic Corporation, 1987.

This Page Intentionally Left Blank

NOTES

NOTES

NOTES

NOTES

NOTES

NOTES

NOTES

NOTES